O'REILLY®



图灵程序设计丛书

第2版



PostgreSQL
即学即用

PostgreSQL: Up and Running

[美] Regina Obe　Leo Hsu 著

丁奇鹏 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍



丁奇鹏

2001年毕业于南京航空航天大学计算数学专业，先后在中兴和华为从事企业自有数据库的研发与技术管理等工作十余年，对关系型数据库的理论体系和内核实现均有深入研究，对于数据库技术在CT领域的应用拥有丰富的实践经验，对PostgreSQL和MySQL等开源数据库也有着浓厚兴趣。现就职于华为固定网络产品线，专注于为华为“云管端”战略体系中的“管道操作系统”进行分布式内存数据库的研发以及性能优化工作。

 图灵程序设计丛书

PostgreSQL即学即用（第2版）

PostgreSQL: Up and Running

Second Edition

[美] Regina Obe Leo Hsu 著
丁奇鹏 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

PostgreSQL即学即用 : 第2版 / (美) 奥贝
(Obe,R.) , (美) 徐 (Hsu,L.) 著 ; 丁奇鹏译. -- 北京:
人民邮电出版社, 2016.1
（图灵程序设计丛书）
ISBN 978-7-115-41128-0

Ⅰ. ①P… Ⅱ. ①奥… ②徐… ③丁… Ⅲ. ①关系数
据库系统 Ⅳ. ①TP311.132.3

中国版本图书馆CIP数据核字(2015)第285900号

内 容 提 要

本书将帮助你理解和使用 PostgreSQL 这一开源数据库系统。你不仅会学到版本 9.2、9.3 和 9.4 中的企业级特性，还会发现 PostgreSQL 不只是个数据库系统，也是一个出色的应用平台。本书通过示例展示了如何实现在其他数据库中难以或无法完成的任务。这一版内容覆盖了 LATERAL 查询、增强的 JSON 支持、物化视图和其他关键话题。

本书适合数据库管理员、后端开发人员以及其他对 PostgreSQL 感兴趣的读者。

◆ 著　[美] Regina Obe Leo Hsu
译　丁奇鹏
责任编辑　岳新欣
执行编辑　赵瑞琳
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：14
字数：330千字　2016年 1 月第 1 版
印数：1 – 3 500册　2016年 1 月北京第 1次印刷

著作权合同登记号　图字：01-2015-6365号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

© 2015 by Regina Obe and Leo Hsu.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2016. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2015。

简体中文版由人民邮电出版社出版，2016。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言 xi

第 1 章　基础知识 1

1.1　如何获得 PostgreSQL 1
1.2　管理工具 1
1.2.1　psql 2
1.2.2　pgAdmin 2
1.2.3　phpPgAdmin 3
1.2.4　Adminer 3
1.3　PostgreSQL 数据库对象 4
1.4　最新版本的 PostgreSQL 中引入的新特性 9
1.4.1　为什么要升级 10
1.4.2　PostgreSQL 9.4 版中引入的新特性 10
1.4.3　PostgreSQL 9.3 版新特性列表 12
1.4.4　PostgreSQL 9.2 版新特性列表 13
1.4.5　PostgreSQL 9.1 版新特性列表 14
1.5　数据库驱动程序 14
1.6　如何获得帮助 15
1.7　PostgreSQL 的主要衍生版本 15

第 2 章　数据库管理 17

2.1　配置文件 17
2.1.1　postgresql.conf 18
2.1.2　pg_hba.conf 21
2.1.3　配置文件的重新加载 23

2.2 连接管理 …… 24
2.3 角色 …… 25
2.3.1 创建可登录角色 …… 26
2.3.2 创建组角色 …… 26
2.4 创建 database …… 28
2.4.1 模板数据库 …… 28
2.4.2 schema 的使用 …… 29
2.5 权限管理 …… 31
2.5.1 权限的类型 …… 31
2.5.2 入门介绍 …… 31
2.5.3 GRANT …… 32
2.5.4 默认权限 …… 33
2.5.5 PostgreSQL 权限体系中一些与众不同的特点 …… 34
2.6 扩展包机制 …… 34
2.6.1 扩展包的安装 …… 36
2.6.2 通用扩展包 …… 38
2.7 备份与恢复 …… 40
2.7.1 使用 pg_dump 进行有选择性的备份 …… 41
2.7.2 使用 pg_dumpall 进行全库备份 …… 42
2.7.3 数据恢复 …… 43
2.8 基于表空间机制进行存储管理 …… 45
2.8.1 表空间的创建 …… 45
2.8.2 在表空间之间迁移对象 …… 45
2.9 禁止的行为 …… 46
2.9.1 切记不要删除 PostgreSQL 系统文件 …… 46
2.9.2 不要把操作系统管理员权限授予 PostgreSQL 的系统账号（postgres） …… 47
2.9.3 不要把 shared_buffers 缓存区设置得过大 …… 47
2.9.4 不要将 PostgreSQL 服务器的侦听端口设为一个已被其他程序占用的端口 …… 47
第 3 章 psql 工具 …… 49
3.1 环境变量 …… 49
3.2 psql 的两种操作模式：交互模式与非交互模式 …… 50
3.3 定制 psql 操作环境 …… 51
3.3.1 自定义 psql 界面提示符 …… 52
3.3.2 语句执行时间统计 …… 53
3.3.3 事务自动提交 …… 53
3.3.4 命令别名 …… 54
3.3.5 取出前面执行过的命令行 …… 54
3.4 psql 使用技巧 …… 55
3.4.1 执行 shell 命令 …… 55
3.4.2 用 watch 命令重复执行语句 …… 55

3.4.3 显示对象信息 ······ 55
3.5 使用 psql 实现数据的导入和导出 ······ 56
3.5.1 使用 psql 进行数据导入 ······ 56
3.5.2 使用 psql 进行数据导出 ······ 58
3.5.3 从外部程序复制数据以及将数据复制到外部程序 ······ 58
3.6 使用 psql 制作简单的报表 ······ 58

第 4 章 pgAdmin 的使用 ······ 61

4.1 pgAdmin 入门 ······ 61
4.1.1 功能概览 ······ 61
4.1.2 如何连接到 PostgreSQL 服务器 ······ 63
4.1.3 pgAdmin 界面导航 ······ 63
4.2 pgAdmin 功能特性介绍 ······ 64
4.2.1 在 pgAdmin 中调用 psql ······ 65
4.2.2 在 pgAdmin 中编辑 postgresql.conf 和 pg_hba.conf 文件 ······ 65
4.2.3 创建数据库资产并设置权限 ······ 66
4.2.4 数据导入和导出 ······ 68
4.2.5 备份与恢复 ······ 69
4.3 pgScript 脚本机制 ······ 72
4.4 以图形化方式解释执行计划 ······ 75
4.5 使用 pgAgent 执行定时任务 ······ 75
4.5.1 安装 pgAgent ······ 76
4.5.2 规划定时任务 ······ 76
4.5.3 一些有用的 pgAgent 相关查询语句 ······ 78

第 5 章 数据类型 ······ 80

5.1 数值类型 ······ 80
5.1.1 serial 类型 ······ 81
5.1.2 生成数组序列的函数 ······ 81
5.2 字符和字符串 ······ 82
5.2.1 字符串函数 ······ 83
5.2.2 将字符串拆分为数组、表或者子字符串 ······ 83
5.2.3 正则表达式和模式匹配 ······ 84
5.3 时间类型 ······ 85
5.3.1 时区详解 ······ 87
5.3.2 日期时间类型的运算符和函数 ······ 89
5.4 数组类型 ······ 91
5.4.1 数组构造函数 ······ 92
5.4.2 引用数组中的元素 ······ 93
5.4.3 数组的拆分与连接 ······ 93
5.4.4 将数组元素展开为记录行 ······ 93

5.5 区间类型 94
5.5.1 离散区间和连续区间 95
5.5.2 原生支持的区间类型 95
5.5.3 定义区间的方法 96
5.5.4 定义含区间类型字段的表 97
5.5.5 适用于区间类型的运算符 97
5.6 JSON 数据类型 98
5.6.1 插入 JSON 数据 98
5.6.2 查询 JSON 数据 99
5.6.3 输出 JSON 数据 101
5.6.4 JSON 类型的二进制版本：jsonb 101
5.7 XML 数据类型 103
5.7.1 插入 XML 数据 103
5.7.2 查询 XML 数据 104
5.8 自定义数据类型和复合数据类型 105
5.8.1 所有表都有一个对应的自定义数据类型 105
5.8.2 构建自定义数据类型 107
5.8.3 为自定义数据类型构建运算符和函数 107

第 6 章 表、约束和索引 109

6.1 表 109
6.1.1 基本的建表操作 109
6.1.2 继承表 110
6.1.3 无日志表 110
6.1.4 TYPE OF 111
6.2 约束机制 112
6.2.1 外键约束 112
6.2.2 唯一性约束 113
6.2.3 check 约束 113
6.2.4 排他性约束 113
6.3 索引 114
6.3.1 PostgreSQL 原生支持的索引类型 115
6.3.2 运算符类 116
6.3.3 函数索引 118
6.3.4 基于部分记录的索引 118
6.3.5 多列索引 119

第 7 章 PostgreSQL 的特色 SQL 语法 121

7.1 视图 121
7.1.1 单表视图 122
7.1.2 使用触发器来更新视图 123

7.1.3 物化视图 ······125
7.2 灵活易用的 PostgreSQL 专有 SQL 语法······127
7.2.1 DISTINCT ON ······127
7.2.2 LIMIT 和 OFFSET 关键字 ······128
7.2.3 简化的类型转换语法 ······128
7.2.4 一次性插入多条记录 ······128
7.2.5 使用 ILIKE 实现不区分大小写的查询······129
7.2.6 可以返回结果集的函数 ······129
7.2.7 限制对继承表的 DELETE、UPDATE、INSERT 操作的影响范围······130
7.2.8 DELETE USING 语法 ······130
7.2.9 将修改影响到的记录行返回给用户 ······130
7.2.10 在查询中使用复合数据类型 ······131
7.2.11 DO······132
7.3 适用于聚合操作的 FILTER 子句 ······133
7.4 窗口函数······135
7.4.1 PARTITION BY 子句 ······136
7.4.2 ORDER BY 子句 ······136
7.5 CTE 表达式 ······138
7.5.1 基本 CTE 用法介绍······139
7.5.2 可写 CTE 用法介绍······140
7.5.3 递归 CTE 用法介绍······140
7.6 LATERAL 横向关联语法 ······141

第 8 章 函数编写······144

8.1 PostgreSQL 函数功能剖析 ······145
8.1.1 函数功能基础知识介绍 ······145
8.1.2 触发器和触发器函数 ······146
8.1.3 聚合操作 ······147
8.1.4 受信与非受信语言 ······149
8.2 使用 SQL 语言来编写函数 ······149
8.2.1 编写基本的 SQL 函数······150
8.2.2 使用 SQL 语言编写聚合函数······151
8.3 使用 PL/pgSQL 语言编写函数······153
8.3.1 编写基础的 PL/pgSQL 函数 ······153
8.3.2 使用 PL/pgSQL 编写触发器函数 ······154
8.4 使用 PL/Python 语言编写函数······155
8.5 使用 PL/V8、PL/CoffeeScript 以及 PL/LiveScript 语言来编写函数 ······157
8.5.1 编写基本的函数 ······159
8.5.2 使用 PL/V8 来编写聚合函数······160

第 9 章　查询性能调优……162
9.1　通过 EXPLAIN 命令查看语句执行计划……162
9.1.1　EXPLAIN 选项……162
9.1.2　运行示例以及输出内容解释……163
9.1.3　图形化展示执行计划……166
9.2　搜集语句的执行统计信息……167
9.3　人工干预规划器生成执行计划的过程……168
9.3.1　策略设置……168
9.3.2　你的索引被用到了吗……169
9.3.3　表的统计信息……170
9.3.4　磁盘页的随机访问成本以及磁盘驱动器的性能……171
9.4　数据缓存机制……172
9.5　编写更好的 SQL 语句……173
9.5.1　在 SELECT 语句中滥用子查询……174
9.5.2　尽量避免使用 SELECT * 语法……176
9.5.3　善用 CASE 语法……177
9.5.4　使用 Filter 语法替代 CASE 语法……178
第 10 章　复制与外部数据……180
10.1　复制功能概览……180
10.1.1　复制功能涉及的术语……181
10.1.2　复制机制的演进……182
10.1.3　第三方复制解决方案……182
10.2　复制环境的搭建……183
10.2.1　主服务器的配置……183
10.2.2　从属服务器的配置……184
10.2.3　启动复制进程……185
10.3　外部数据封装器……186
10.3.1　查询平面文件……186
10.3.2　以不规则数组的形式查询不规范的平面文件……187
10.3.3　查询其他 PostgreSQL 服务实例上的数据……188
10.3.4　查询非传统数据源……190
附录 A　PostgreSQL 的安装……192
附录 B　PostgreSQL 自带的命令行工具……196
作者简介……204
封面介绍……204

前言

PostgreSQL（http://www.postgresql.org/）是一个开源的关系型数据库管理系统，最初源于加州大学伯克利分校的一个研究项目。该系统最早是基于 BSD 许可证发布的，但目前已改为使用 PostgreSQL 许可证（简称 TPL）发布。事实上这两种许可证无论从哪方面看都没有区别。PostgreSQL 的悠久历史可追溯到 1985 年。

PostgreSQL 拥有诸多企业级特性，比如支持窗口函数（用户可以自定义聚合函数并当作窗口函数使用）、普通 CTE 表达式、递归 CTE 表达式以及流式复制等。这些特性在 Oracle、SQL Server、DB2 等较新版本的商用数据库中很常见，但在开源数据库界却几乎没有。另外，PostgreSQL 有一点与众不同，它可以在不用重编译任何代码的情况下轻松实现系统功能的扩展。PostgreSQL 不但支持众多高级特性，而且性能也很好，在很多应用场景下其性能甚至可以超越包括商用数据库在内的大多数数据库。

本书将介绍 PostgreSQL 的诸多高级特性，其中有的特性是 ANSI SQL 标准中所规定的，而有的特性是 PostgreSQL 自己独创的。如果你当前正在使用 PostgreSQL，又或者以前曾用过但了解程度一般，那么通过本书可以学到之前可能错过的一些功能“遗珠”，还可以了解到最新几个版本中引入的新特性。本书适合对关系型数据库有一定使用经验的读者，但不要求使用过 PostgreSQL。书中将对比 PostgreSQL 与其他数据库处理同一任务的机制，同时也将展示只有 PostgreSQL 才支持的一些“高大上”功能，这些功能在别的数据库中要么实现起来很困难，要么根本不可能实现。如果你完全未使用过数据库，通过本书也可以学到 PostgreSQL 的功能和使用方法。不过，鉴于本书的定位，书中不会过多介绍关于 SQL 或者关系型数据库理论方面的基础知识，我们建议你阅读其他相关书籍来了解这些内容。

本书主要介绍 PostgreSQL 9.2、9.3 和 9.4 版，但也会覆盖一些在更早版本中已支持的高级特性。

本书读者

我们希望本书对数据库行业现有从业人员以及刚刚开始从事数据库领域工作的读者能够有所帮助。具体来说，本书面向的读者群如下。

- 如果你正在学习关系型数据库，那么我们希望本书能够对你有所帮助，并希望你在将来的职业生涯中青睐于 PostgreSQL。为了降低难度，我们在本书第 2 版中对许多专题进行了拓展介绍，并尽可能提供了入门级的例子。
- 如果你当前已经是 PostgreSQL 的用户或者 DBA，那么我们希望此书能让你的工作更加得心应手。书中会有你已经熟悉的内容，但也一定会有你不熟悉的一些技巧，以及新版本中引入的新特性，如果善加利用，必定会提高你的工作效率。好吧，如果你真的很出色并且对书中内容均已了解，那么本书对你来说依然有价值，为什么呢？因为它比官方的 PostgreSQL 手册要轻上 20 倍，最起码是便于携带了。
- 如果你还没接触过 PostgreSQL，那么本书除了能够向你介绍 PostgreSQL 知识外，还将扮演你身边的 PostgreSQL“布道师”的角色，这位“布道师”将向你证明：多绑在商业数据库上一天，你就会被那些厂商掏走更多的钱；多用那些“烂”数据库一天，你的系统就不得不多做一天功能上的妥协。

如果你的工作与数据库领域甚至是 IT 界毫无关系，又或者你刚刚幼儿园毕业，那么能否购买本书呢？答案依然是可以的！因为封面上可爱的象鼩鼠图片就已经让本书物有所值了。

PostgreSQL的特别之处以及选用理由

PostgreSQL 之所以特别，是因为它不仅仅是一个数据库，还是一个功能强大的应用开发平台。

PostgreSQL 支持用多种编程语言编写存储过程和函数。除了系统自带的编程语言外，还可以通过安装语言扩展包来支持新的语言。内置的基础语言包括 SQL 和 PL/pgSQL，通过安装扩展包还可以支持 PL/Perl、PL/Python、PL/V8（又称为 PL/JavaScript）以及 PL/R 等。前述语言的安装包在 PostgreSQL 发行版中大多都已自带，需要时安装一下扩展包即可使用。这种支持多语言的能力是非常有价值的，因为每种语言的特点不同，有的语言适合解决特定领域内的问题，有的语言过程化特性更丰富，有的语言语法特性更强大，那么开发人员就可以根据待解决问题的特点来选择最合适的语言。比如可以通过使用 R 语言的统计和图形函数以及 R 语言中简洁的专业表达式来解决统计领域的问题，可以通过 Python 来调用 Web 服务，也可以编写 map reduce 函数并在 SQL 语句中调用。

用户甚至可以写一个聚合函数并嵌入到上述这些语言中，这样就可以把 SQL 语言的聚合运算能力与上述语言本身的能力相结合，从而拓展了这些语言的能力范围。此外，PostgreSQL 还支持调用 C 语言写的函数，与调用上述其他语言写的函数没有区别。可以在

同一个 SQL 语句中调用分别由不同语言编写的多个函数。甚至可以仅使用 SQL 语言（即无过程化能力的纯 SQL）来创建一个用户自定义聚合函数。在 MySQL 和 SQL Server 中，用户自定义聚合函数是需要对数据库软件本身进行重编译才能做到的，但在 PostgreSQL 中却不需要。简而言之，PostgreSQL 对多语言的支持是极其灵活和强大的，开发人员不但能够为不同任务选择不同的编程语言，也能够为同一任务的不同子任务选用不同的语言。另外，在其他绝大多数数据库都不允许使用 SQL 的场景下，PostgreSQL 也允许你使用 SQL。PostgreSQL 中不用编译任何代码就可以创建非常复杂的函数。

PostgreSQL 支持非常强大的用户自定义数据类型功能，不但使用方法简单，而且性能也超越了绝大多数其他关系型数据库。在用户自定义数据类型方面，与 PostgreSQL 实力最接近的对手只有 Oracle。用户可以在 PostgreSQL 中定义新的数据类型，然后就可以将该数据类型用作表列类型。每种数据类型都有伴随的数组类型，这样可以将某个类型的数组存储到某个数据列中，或者在 SQL 语句中使用该数组。另外还可以为新增的数据类型定义相应的运算符、函数和索引绑定来与其协同工作。很多 PostgreSQL 的第三方扩展包就利用该自定义数据类型能力来优化性能，或者通过添加支持某个领域专用的特殊 SQL 语法来让业务代码更简洁和易于维护，或者实现一些在别的数据库中完全不可能实现的功能。

如果不需要自定义数据类型和相应的函数，那么系统自带的数据类型也是种类繁多的，比如 `json`（在 9.2 版中引入），另外还有很多数据类型扩展包可供选择。很多这类扩展包是 PostgreSQL 发行版中自带的。从 PostgreSQL 9.1 开始引入了一种新的语法：`CREATE EXTENSION`。该语法仅通过一条 SQL 语句就实现了扩展包的安装。如果需要在某个数据库中使用某个扩展包的功能，则在该数据库中安装该扩展包即可。通过 `CREATE EXTENSION` 语法，可以安装前述任何一种过程式语言（简称 PL），以及使用比较广泛的数据类型及其相应的函数和运算符，比如 `hstore` 键 – 值存储、`ltree` 层次化存储、PostGIS 地理空间扩展以及数不胜数的其他扩展。举个例子，如果想加载 `hstore`，只需执行下面这个命令即可：

```
CREATE EXTENSION hstore;
```

此外，有一条 SQL 命令可以列出所有可用的以及已安装的扩展包（详见 2.6 节）。

前面提到了 PostgreSQL 支持各种扩展，也提到了它对多种编程语言的支持，但你可能对这些都不感兴趣。你可能觉得："哦，支持 Python，还支持 Perl……可那又怎么样？能不能说点我不知道的？"别着急，当我们继续往后深入学习的时候，你一定能够不时地体会到"哇，这个功能太牛了！"这样的惊喜，而这样的惊喜在我们多年使用 PostgreSQL 的过程中已经体会过太多次。PostgreSQL 的每次升级都会为用户提供新的特性，这些特性包括易用性的升级、性能的提升，以及对于关系型数据库功能极限的不断超越。到最后你会发现：我为什么还要使用别家的数据库？ PostgreSQL 已经提供了我所需要的一切功能了啊，而且还是免费的！你不再需要去阅读那些商业数据库附带的密密麻麻的授权条款，来了解

在一个 8 核服务器上支持甲特性、乙特性和丙特性所需要的费用是多少，也不需要了解如果服务器 CPU 从 8 核升级到 16 核后要再为许可证加多少钱。

此外，PostgreSQL 在其支持的所有平台上的功能表现都很一致。因此你根本不需要担心你的客户要求支持哪种操作系统，Unix、Linux、Mac OS X、Windows，PostgreSQL 全都支持。PostgreSQL 官方站点提供各种操作系统下的二进制安装包下载，当然你也可以自行编译安装。

不适用PostgreSQL的场景

PostgreSQL 从一开始就被设计为一个通用的事务型数据库。很多人把它用在小型的桌面应用程序中，就类似一个 SQL Server Express 免费版或者是 Oracle Express 免费版。但这种用法存在的问题是：作为一个独立的数据库系统，PostgreSQL 自身会进行全面的安全管理（比如用户登录机制，这个机制在嵌入式场景下是不需要的），这些都是要耗费性能的，但 PostgreSQL 又无法取消安全机制并将其交由上层应用去管理，因此单用户应用场景下 PostgreSQL 可能不是最好的选择。此种场景下 SQLite 或者 Firebird 是更合适的选择，因为用户权限管理、安全检查和 DB 操作日志功能都是由上层应用自己完成的。

令人遗憾的一个事实是，很多共享主机（多个租户共享同一个操作系统）供应商并不支持预安装 PostgreSQL，或者只支持安装一个很陈旧的版本。因此，如果在使用共享主机服务，你可能就不得不使用 MySQL。本书第 1 版出版以后，这个情况有了很大改善。目前虚拟专用主机和云服务器（每个租户独享一个操作系统，多租户之间互不干扰）的租用价格已经趋于合理，而且会越来越便宜。与共享主机相比，其价格不会高出很多，而且可以在上面安装任何你希望安装的软件。因此选择专用的云主机服务会是比较明智的做法，因为你可以安装 PostgreSQL 的最新稳定版，而无需受制于与其他租户共享主机时的种种限制。此外，主流的 PaaS 平台均已支持 PostgreSQL，而且一般都支持最新的发行版。这些主流平台包括：SalesForce Heroku PostgreSQL、Engine Yard、Red Hat OpenShift，以及 Amazon RDS for PostgreSQL。

PostgreSQL 的功能极其强大，强大到“令人生畏”。它绝对不是一套仅仅能做些数据存储的小软件，它是如同一只聪慧的大象般智能又强大的大型系统。如果你所需要的仅仅是一个键 – 值存储，或者一个只要能装数据就行的小软件，那么 PostgreSQL 必定会远比你的期望强大百倍。

如何获取本书使用的数据和示例代码

可从本书的官方链接（http://www.postgresonline.com/downloads/postgresql_book_2e.zip）获取。如果发现所提供的数据有误，请将相关信息发布到本书的勘误页面（http://www.oreilly.com/catalog/errata.csp?isbn=0636920032144）。

关于PostgreSQL的更多信息

本书致力于展示 PostgreSQL 区别于其他数据库的独特功能，以及如何使用这些功能来解决现实世界的问题。你将学到如何使用一些你在数据库领域前所未见的方法来解决问题。除了那些“高大上”的内容，我们也会向你展示如何处理一些基本的任务，比如数据库管理、权限设置、性能问题定位、性能调优、使用不同的桌面工具连接到数据库、命令行操作以及开发工具的使用，等等。

PostgreSQL 有一套内容丰富的在线文档。本书不会重复这些文档中已提供的信息，而是会鼓励你去探索博大精深的 PostgreSQL 中的未知领域。官方手册（http://www.postgresql.org/docs/manuals）目前包含 2250 多页的内容，同时提供 HTML 和 PDF 两种格式。此外，如果你需要的话，最近几个版本的官方手册还提供纸质印刷版。由于其规模庞大，内容丰富，纸质版通常是分成 3 到 4 册提供的。

其他可用的 PostgreSQL 资源还包括：

- Planet PostgreSQL（http://planet.postgresql.org）是 PostgreSQL 技术博客文章的汇聚站点，其中包含从 PostgreSQL 核心开发人员到普通用户编写的各类文章，包括新特性演示以及对现有功能的使用说明；
- PostgreSQL Wiki（https://wiki.postgresql.org）提供对 PostgreSQL 各个方面的使用技巧说明，以及从其他数据库移植到 PostgreSQL 的方法；
- PostgreSQL Books（http://www.postgresql.org/docs/books/）提供有关 PostgreSQL 的书籍列表信息；
- PostGIS in Action Books（http://www.postgis.us/）是我们已出版的关于 PostGIS 的书籍的官方站点。

代码与输出格式

对于括号中的内容，我们一般会将左括号与之前的内容放置于同一行，右括号单独放置一行，以便于纵栏印刷。格式如下：

```
function ( Welcome to PostgreSQL
);
```

为节省版面，我们还移除了命令行执行输出结果中无意义的空格，因此如果发现实际输出结果的格式与书中提供的不一致，请不用担心，这是正常的。

请注意，虽然我们建议在编码时逗号后要加一个空格，但本书中的确有些地方因版面宽度的关系而去掉了这种空格。

PostgreSQL 的 SQL 解释器会将语句中的制表符、换行符和回车符当作空格处理。在我们

提供的示例代码中，一般会使用空格而不是制表符作为缩进符。请确保使用的代码编辑器不会自动将制表符、换行符和回车符删除，或者把它们转换为空格以外的什么字符，否则会导致问题。

如果在执行示例代码时遇到了问题，请检查确认你复制过来的代码与我们提供的原始代码是否一致。

注意有些示例适用于 Linux，而有些适用于 Windows。例如某些关于外部数据封装器的例子中要求使用带完整路径的文件名，你会看到示例代码中有类似于 `/postgresql_book/somefile.csv` 这样的路径,这指的是 Linux 服务器根目录下的路径。如果使用的是 Windows 环境，那么需要加上驱动器符，因此路径要改为：`C:/postgresql_book/somefile.csv`。请注意：即使是在 Windows 上，路径中也应该使用 Linux 的路径分隔符 `/`，而不是 Windows 传统的 `\`。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 斜体等宽字体（*`constant width bold`*）
  表示应替换成用户提供的值或由上下文决定的值。



该图标表示提示或建议。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 http://www.postgresonline.com/downloads/postgresql_book_2e.zip 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*PostgreSQL: Up and Running, Second Edition* by Regina Obe and Leo Hsu (O'Reilly). Copyright 2015 Regina Obe and Leo Hsu, 978-1-4493-7319-1."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920032144.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

第 1 章

基础知识

本章将带你开始 PostgreSQL 的探索之旅。首先将介绍如何下载和安装 PostgreSQL，然后会讲到一些必备的管理工具和 PostgreSQL 术语。本书写作之时，PostgreSQL 9.4 正在等待发布，我们将重点介绍该版本的一些新特性。在本章的末尾，我们将给出一些帮助资源列表，在遇到问题时你可以从中获得帮助。

1.1　如何获得PostgreSQL

若干年前，你只能通过手动编译源码的方式来安装 PostgreSQL。还好那种痛苦的时代已经一去不复返了。当然，现在依然可以通过编译源码来安装，但大多数用户会使用制作好的安装包来安装，只需敲击几下键盘和鼠标就可以了。

如果你是首次安装 PostgreSQL，那么应该选择适用于你的操作系统平台的最新稳定版发行包。PostgreSQL 官方站点的核心发布页面上维护了一个列表（http://www.postgresql.org/download），记录了适用于各操作系统的二进制包的下载地址。在附录 A 中，你将会看到安装指导和一些特殊定制过的版本的下载链接地址。

1.2　管理工具

PostgreSQL 常用的管理工具有四种：psql、pgAdmin、phpPgAdmin 和 Adminer。PostgreSQL 的核心开发团队维护着前三种，因此它们一般会随着 PostgreSQL 的版本发布而同步更新。Adminer 并非 PostgreSQL 的专用管理工具，它支持管理多种类型的关系型数据库，包括 SQLite、MySQL、SQL Server 和 Oracle 等。除了我们刚刚提到的这四种以外还有大量优秀

的管理工具，开源的和商业的都有。

1.2.1 psql

psql 是一种用于执行查询的命令行工具，每个 PostgreSQL 发行版中都自带 psql。它有一些独特的功能，比如导入或者导出基于分隔符（分隔符可以是逗号或者制表符等字符）格式的平面数据文件，以及生成简易的 HTML 格式报表等。psql 是 PostgreSQL 从诞生之初就一直附带的命令行工具，它是很多资深 PostgreSQL 专家日常操作工具的不二之选，非常适用于只有控制台字符界面而无图形用户界面的使用场景。另外在通过 shell 脚本执行数据库操作时，psql 也是必备工具。不过新用户一般更喜欢使用图形界面工具，而且也无法理解为什么“老”一代人会对命令行方式那么执着。

1.2.2 pgAdmin

pgAdmin（www.pgadmin.org）是一种广泛使用的开源 PostgreSQL 图形界面管理工具。如果你的 PostgreSQL 安装包里没有附带此工具，请从其官网单独下载安装。

pgAdmin 运行于图形化桌面环境下，可以同时连接到多个 PostgreSQL 服务器上，这些服务器可以是安装在任意操作系统平台上的任意 PostgreSQL 版本。

即使你的数据库安装在只有字符控制台界面的 Linux 服务器上，只要你在本地工作站上安装了 pgAdmin，也可以用这种强大的图形化工具对其进行管理。

图 1-1 是 pgAdmin 的界面示意图。

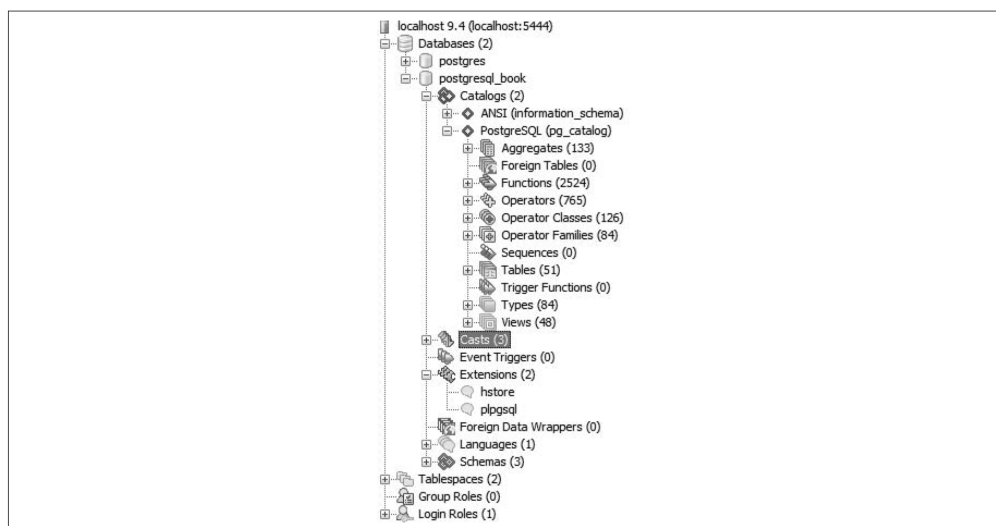


图 1-1：pgAdmin

如果你对 PostgreSQL 还不太熟悉，那么 pgAdmin 毫无疑问将是你开始 PostgreSQL 学习之旅的最佳入口。只需要在主界面上摸索一下，你就可以对 PostgreSQL 的丰富功能一览无遗。如果你正打算逃离 SQL Server 阵营，并且习惯于 SQL Server 的 Management Studio，那么很快就能适应 pgAdmin。

1.2.3 phpPgAdmin

phpPgAdmin（https://github.com/phppgadmin/phppgadmin）是一种免费的基于 Web 页面的管理工具，其界面如图 1-2 所示。它是从流行的 MySQL 管理工具 phpMyAdmin 移植而来的，二者的差别主要在于 phpPgAdmin 新增了对 schema、过程化语言、类型转换器、运算符等对象的管理功能。如果你对 phpMyAdmin 很熟悉，会发现 phpPgAdmin 的界面风格与其完全类似。

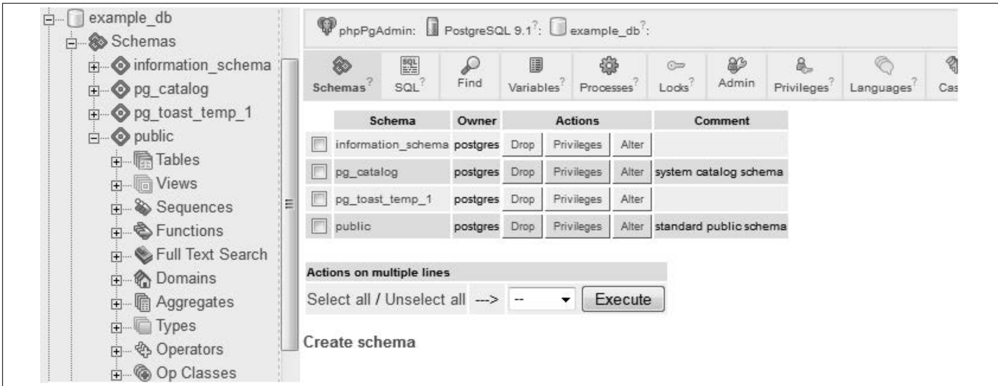


图 1-2：phpPgAdmin

1.2.4 Adminer

如果你正在寻找一款除了能够管理 PostgreSQL，还能管理别的数据库的整合型工具，那么 Adminer（http://www.adminer.org/）将是你合适的选择。Adminer 是一款轻量级的开源 PHP 应用程序，可以在同一套图形界面上管理 PostgreSQL、MySQL、SQLite、SQL Server 以及 Oracle 等多种数据库。

Adminer 有一种独特的功能让我们印象深刻：它能够以图形化方式展示数据库中的对象，并将外键约束关系以连接线的方式展示出来。另外，整个 Adminer 程序的本体仅包含一个 PHP 文件，非常简洁，这可以大大减少你安装部署时的麻烦。

图 1-3 中，左侧是登录屏幕的截图，右侧是表间关系图形化后呈现的效果。很多用户会因为登录屏幕上没有填写端口号的地方而感到困惑。如果 PostgreSQL 使用标准的 5432 侦听端口，那么登录时不填也没问题；但如果不是，那么就需要在服务器名称后面加上端口

号，注意用冒号分隔主机名和端口号，如图 1-3 所示。

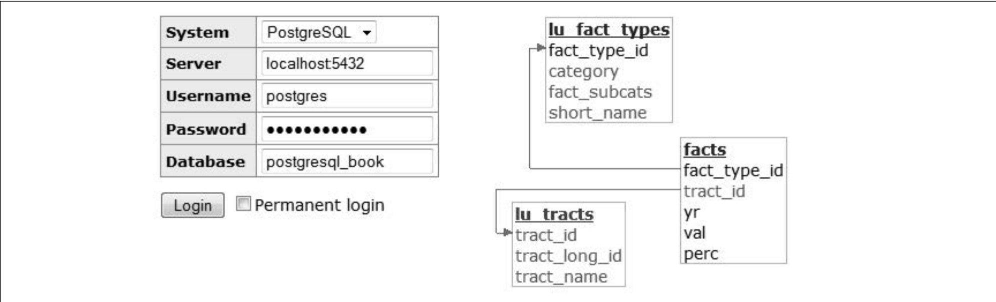


图 1-3：Adminer

对于简单的查询和修改操作来说，Adminer 的功能是足够的。但为了支持多种数据库，Adminer 的功能体系已经被裁剪成了各数据库均支持的最小公共集合，因此你无法实现 PostgreSQL 所特有的一些操作，比如创建新用户、授予权限或者是查询当前权限列表等。如果你是 DBA，那么建议使用 pgAdmin，当然也可以安装一套 Adminer 以备不时之需。

1.3 PostgreSQL数据库对象

假设你现在已经安装好了 PostgreSQL，请启动并连接好 pgAdmin，然后点开左侧的目录树，此时展现在你面前的是一堆令人眼花缭乱的数据库对象，有些你可能很熟悉，有些则可能闻所未闻。PostgreSQL 对象类型的数量超过了绝大多数关系型数据库（这还是在未安装任何扩展包的情况下）。这些对象中，有许多你可能永远都不会用到，但如果你发现业务上需要实现一种新的对象类型，那么一般来说你要实现的东西在那一堆眼花缭乱的对象中已经有前人实现过了，所以只需要正确选用即可。本书不会介绍 PostgreSQL 以标准方式安装完毕后所提供的所有对象类型，因为 PostgreSQL 引入新特性的速度惊人，任何一本书都不可能全面覆盖所有对象类型。因此我们仅讨论你有必要了解的那些对象类型。

- 服务

  在大多数操作系统上，PostgreSQL 是作为一种服务（或者叫守护进程）安装的。多个 PostgreSQL 服务可以运行于同一台物理服务器上，但它们的侦听端口不能重复，也不能共享同一个数据存储目录。本书中，server（服务器）和 service（服务）这两个术语可互换使用，因为大多数人在一台物理服务器上只会安装一个服务。

- database[1]

  每个 PostgreSQL 服务可以包含多个独立的 database。

注 1：database 一词含义宽泛，既可表示广义的数据库系统，又可以表示某些特定数据库系统中的某一级数据存储单位，如表述不当极易给读者造成混淆。因此本书中会区别使用，表示广义的数据库系统时，用中文“数据库”；表示狭义的数据存储单位时，用英文“database”。——译者注

- schema[2]

  ANSI SQL 标准中对 schema 有着明确的定义，database 的下一层逻辑结构就是 schema。如果把 database 比作一个国家，那么 schema 就是一些独立的州（或者是省、府、辖区等，具体取决于各国的实际情况）。大多数对象是隶属于某个 schema 的，然后 schema 又隶属于某个 database。在创建一个新的 database 时，PostgreSQL 会自动为其创建一个名为 `public` 的 schema。如果未设置 `search_path` 变量（后续会介绍该变量的含义），那么 PostgreSQL 会将你创建的所有对象默认放入 `public` schema 中。如果表的数量较少，这是没问题的，但如果你有几千张表，那么我们还是建议你将它们分门别类放入不同的 schema 中。

- catalog[3]

  catalog 是系统级的 schema，用于存储系统函数和系统元数据。每个 database 创建好以后默认都会含有两个 catalog：一个名为 `pg_catalog`，用于存储 PostgreSQL 系统自带的函数、表、系统视图、数据类型转换器以及数据类型定义等元数据；另一个是 `information_schema`，用于存储 ANSI 标准中所要求提供的元数据查询视图，这些视图遵从 ANSI SQL 标准的要求，以指定的格式向外界提供 PostgreSQL 元数据信息。

  一直以来，PostgreSQL 数据库的发展都严格地遵循着其“自由与开放”的核心理念。如果你足够了解这款数据库，会发现它几乎是一种可以“自我生长”的数据库。比如，它所有的核心设置都保存在系统表中，用户可以不受限地查看和修改这些数据，这为 PostgreSQL 提供了远超任何一种商业数据库的巨大灵活性（不过从另一个角度看，将这种灵活性称为“可破坏性”也未尝不可）。只要仔细地研究一下 `pg_catalog`，你就可以了解到 PostgreSQL 这样一个庞大的系统是如何基于各种部件构建起来的。如果你有超级用户权限，那么可以直接修改 `pg_catalog` 的内容（当然，如果改得不对，那你的行为就跟搞破坏没什么两样了）。

  `Information_schema` catalog 在 MySQL 和 SQL Server 中也有。PostgreSQL `Information_schema` 中最常用的视图一般有以下几个：`columns` 视图，列出了数据库中的所有表列；`tables` 视图，列出了数据库中的所有表（包括视图）；`views` 视图，列出了所有视图以及用于构建或重新构建该视图的关联 SQL。同样，在 MySQL 和 SQL Server 中也有这些视图，不过它们所含的列没有 PostgreSQL 那么多。PostgreSQL 另外添加了几个用于描述自定义数据类型列的列，比如 `columns.udt_name`。

注 2：数据库业界对于 schema 有多种译法：纲要、模式、方案，等等。但各种译法都不能准确直观地表达出其原本的含义，即位于一个独立命名空间内的一组相关数据库对象的集合，因此前述译法从来没有一种成为主流。一般业界人员都直接使用英文 schema。考虑到这个情况，为防止初级用户理解困难，我们也按照业界习惯直接使用英文原名。——译者注

注 3：catalog 的译法与 schema 存在相同的问题，翻译为“目录”后并不能让读者准确地理解其原意，反而容易造成混淆，因此还是沿用英文原名。——译者注

尽管 `columns`、`tables`、`views` 这三个元数据视图本身也是标准的 PostgreSQL 视图，但由于其身份的特殊性，pgAdmin 界面中还是把它们放在了 information_schema → Catalog Objects 分支下。

- 变量

  变量是 PostgreSQL 统一配置机制（GUC）的一部分，是可以在多个级别进行设置的各种选项，这些级别包括服务级、database 级以及其他级别。很多人容易在 `search_path` 这个选项上犯错，该选项的工作机制是：如果在 `search_path` 中指定了 schema 的名称，那么该 schema 资产在使用时就无需显式指定其 schema 名，系统会按照 `search_path` 中登记的 schema 名按顺序逐个搜索。2.4.2 节会对 `search_path` 进行详细讨论。

- 扩展包

  PostgreSQL 9.1 版中引入了对于扩展包机制的支持。开发人员可以通过该机制将一组相关的函数、数据类型、数据类型转换器、用户自定义索引、表以及 GUC 等对象打包成一个功能扩展包，该扩展包可以整体安装或者整体删除。扩展包在概念上与 Oracle 的 package 类似，一般推荐使用该机制来为数据库提供功能扩展。关于扩展包的具体安装步骤，请参考开发手册。一般来说需要先将扩展包的二进制安装包和脚本复制到服务器，然后再为需要该扩展包功能的 database 单独安装。

  假设你需要用到某个扩展包的功能，那么仅需将其安装到对应的 database 中即可，而不必为当前数据库系统中的每个 database 都安装一遍。比如需要对某个 database 中的数据进行高级文本搜索，那么单独为该 database 安装 `fuzzystrmatch` 扩展包即可。安装扩展包时可以指定装到哪个 schema，若不指定则默认会装到 `public` schema 中。我们不建议这么做，因为这会导致 `public` schema 变得庞大复杂且难以管理，尤其是如果你将自己的数据库对象也都存入 `public` schema 中，那么情况会变得更糟糕。我们建议你创建一个独立的 schema 用于存放所有扩展包的对象，甚至为规模较大的扩展包单独创建一个 schema。为避免出现找不到新增扩展包对象的问题，请将这些新增的 schema 名称加入 `search_path` 变量中，这样就可以直接使用扩展包的功能而无需关注它到底装到了哪个 schema 中。也有一些扩展包明确要求必须安装到某个 schema 下，这种情况下你就不能自行指定了。例如有很多语言扩展包，比如 plv8，就要求必须安装到 `pg_catalog` 中。

- 表

  任何一个数据库中，表都是最核心和最“忙碌”的对象类型。在 PostgreSQL 中，表首先属于某个 schema，而 schema 又属于某个 database，这样就构成了一种三级存储结构。

  PostgreSQL 的表支持两种很强大的功能。第一种是表继承，即一张表可以有父表和子表。这种层次化的结构可以极大地简化数据库设计，还可以为你省掉大量的重复查询代码。我们将在本书后面的示例 6-2 中介绍表继承机制。

第二种是创建一张表的同时，系统会自动为此表创建一种对应的自定义数据类型。换句话说，你可以将某个完整的数据结构定义为一个表，然后将该表用作另一个表中的一个列。关于这种复合数据类型的更多细节，请参见 5.8 节。

- *外部表和外部数据封装器*

  *外部表*的首次亮相是在 9.1 版中。它们是一些虚拟表，通过它们可以直接在本地数据库中访问来自外部数据源的数据。只要数据映射关系配置正确，那么外部表的用法就与普通表没有任何区别。外部表支持映射到以下类型的数据源：CSV 文件、另一个服务器上的 PostgreSQL 表、SQL Server 或 Oracle 这些异构数据库中的表、Redis 这样的 NoSQL 数据库或者甚至像 Twitter 或 Salesforce 这样的 Web 服务。外部表映射关系的建立是通过配置*外部数据封装器*（Foreign Data Wrapper，FDW）实现的。FDW 是 PostgreSQL 和外部数据源之间的一架"魔法桥"，可实现两边数据的互联互通。其内部实现机制遵循 SQL 标准中的 MED（Management of External Data）规范，更多细节请参考维基百科上关于 MED 的描述（http://en.wikipedia.org/wiki/SQL/MED）。

  许多编程人员已经为当下绝大部分流行的数据源开发了 FDW 并已免费共享出来。你也可以通过创建自己的 FDW 来练习。（我们建议你一旦成功了也公布出来，这样整个社区都可以分享你的劳动成果。）FDW 是通过扩展包机制实现的，装好以后在 pgAdmin 界面上名为 Foreign Data Wrapper 的目录节点下能看到它。

- *表空间*

  表空间是用于存储数据的物理空间。PostgreSQL 将用于物理存储的表空间和用于逻辑存储的 schema 分开管理，二者之间并无耦合关系。这样就很容易在不影响业务应用逻辑的情况下，将 database 甚至是单张表和索引在不同的物理驱动器之间进行移动。

- *视图*

  大多数关系型数据库都支持视图，通过视图可以大大简化复杂的查询逻辑，另外也可以通过对视图执行更新操作来修改其基表数据。PostgreSQL 也不例外，从 9.3 版开始支持对基于单表的视图直接使用 SQL 进行更新操作，这样就不需要再写额外的规则或者触发器来实现对此类简单视图的更新。但对于包含更复杂逻辑的视图，或者是基于多张表的视图，对其进行数据更新操作时仍需编写规则或者触发器。9.3 版还引入了对物化视图的支持，该机制通过对视图数据进行缓存来实现对常用查询的加速。更多细节请参见 7.1.3 节。

- *函数*

  PostgreSQL 中函数执行后的返回结果可以是一个标量值或几个记录集。可以在函数中对表数据进行修改，其他数据库对于这种会修改表记录的函数一般称为存储过程。

- *内置编程语言*

  函数是以过程化语言（Procedural Language，PL）编写的。PostgreSQL 默认支持三

种内置编程语言：SQL、PL/pgSQL 以及 C 语言。可以通过 `CREATE EXTENSION` 或者 `CREATE PRODCEDURAL LANGUAGE` 命令来安装其他语言包。目前较常用的语言有 Python、JavaScript、Perl 以及 R。第 8 章中有大量的相关示例。

- *运算符*

  运算符本质上是符号化的已命名函数（例如 `=`、`&&` 等），它需要一个或者两个实参（argument），底层有一个相应的函数来实现其运算逻辑。PostgreSQL 支持自定义运算符。如果你定义了自己的数据类型，那么可定义一些运算符来与之配合工作。比如可以为自定义数据类型定义 `=` 运算符。你甚至可以为两个完全不同的数据类型定义一个运算符，来对其进行比较运算。

- *数据类型（或者仅仅类型）*

  每种数据库产品都会支持一系列的数据类型，比如整型、字符型、数组，等等。除前述常见类型外，PostgreSQL 还支持*复合数据类型*，这种类型可以是多种数据类型的一个组合，比如虚数、极坐标、张量都是复合数据类型。如果你定义了自己的复合数据类型，那么可以创建一组函数和运算符来配合它工作，可以做得很专业，很复杂，很“高端”，比如自定义实现 `div`（散度运算）、`grad`（梯度运算）和 `curls`（旋度运算）等。哪位读者若有兴趣，可以试一下。

- *数据类型转换器*

  cast 是数据类型转换器，就是将一种数据类型转换为另一种类型的工具。转换器在其底层其实是通过调用转换函数来实现真正的转换逻辑的。PostgreSQL 的独到之处在于支持用户自定义转换器，这样就可以改变系统默认的转换行为。例如，如果你需要把邮政编码（美国的邮政编码是一个 5 位的整数）从 `integer` 转换为 `character`，那么可以自定义一个支持“数字不足 5 位则前面自动补 0”规则的转换器。转换器可以被*隐式*调用也可以被*显式*调用。隐式转换是系统自动执行的，一般来说，将一种特定数据类型转为更通用的数据类型（比如数字转换为字符串）时就会发生隐式类型转换。如果进行隐式转换时系统找不到合适的转换器，你就必须显式执行转换动作。

- *序列*

  序列控制 serial 数据类型的自动递增。在 PostgreSQL 中定义 serial 列时，PostgreSQL 会自动创建序列，但你很容易更改初始值、增量和下一个值。因为序列是独立对象，所以多个表可以使用同一个序列对象。这样你可以创建跨越多个表的独特键值。SQL Server 和 Oracle 也都有序列对象，但必须手动创建。

- *行或记录*

  本书中，“行”和“记录”这两个术语可互换使用。在 PostgreSQL 中，“记录”这个概念可以脱离表而独立存在。当你在函数或者在 SQL 语句中使用记录构造函数语法（语法类似于：`SELECT ROW(1,2.5,'this is a test')`）时，会对这一点有深刻体会。

- 触发器

  绝大多数企业级数据库都支持触发器机制，该机制可以实现对数据修改事件的捕获，并在之后触发用户自定义的操作行为。触发器的触发时机是可设置的，可以是语句级触发或者记录级触发，可以是修改前触发也可以是修改后触发。

  PostgreSQL 的触发器技术正在快速的演进之中。9.0 版引入了对 `WITH` 子句的支持，通过它可以实现带条件的记录级触发，即只有当某条记录符合指定的 `WHEN` 条件时，触发器才会被调用。9.0 版还引入了 `UPDATE OF` 子句，通过它可以指定要监控哪些列的更改。当列更改时，就会触发触发器，详情请参见第 8 章中的示例 8-11。在 9.1 版中，视图中的数据更改可以触发触发器。在 9.3 版中，数据定义语言（DDL）事件可以触发触发器。目前支持触发器的 DDL 命令列表请参见官方手册中"触发器触发时机一览表"（http://www.postgresql.org/docs/9.4/interactive/event-trigger-matrix.html）。在 9.4 版中，针对外部表的触发器也获得了支持。请参考官方文档中"创建触发器"（http://www.postgresql.org/docs/current/interactive/sql-createtrigger.html）这一节的内容以获取更多信息。

- 规则

  规则是一种能够将一种动作替换为另一种动作的机制。PostgreSQL 内部就是使用此机制来实现视图的。比如我们定义了这样一个视图：

  ```
  CREATE VIEW vw_pupils AS SELECT * FROM pupils WHERE active;
  ```

  实际上 PostgreSQL 在后台自动创建了一个 `INSTEAD OF SELECT` 类型的规则，其内容是：当查询名为 `vw_pupils` 的表时，自动重定向为查询 `pupils` 表中 `active` 字段值为 `true` 的记录。

  规则还可以用于替代一些特定的简单触发器。通常情况下，在对记录行进行更新 / 插入 / 删除操作时会调用触发器。规则却不一样，它的运作机理是修改用户原本的行为逻辑（也就是你执行的 SQL 语句本身），或者是在用户原有逻辑的基础上额外附加一些 SQL 逻辑。相比触发器而言，这种整体取而代之的方式避免了针对每条记录都调用一次触发器所造成的负担。一般来说，如果你希望在数据修改时加载自定义逻辑，我们还是推荐使用触发器而不是规则。很多 PostgreSQL 用户认为规则是过时的技术，因为出问题时很难诊断，而且规则只支持用 SQL 语法来编写，PostgreSQL 所支持的其他编程语言则无用武之地。

1.4 最新版本的PostgreSQL中引入的新特性

PostgreSQL 的版本发布是很有规律的，每年的 9 月份会发布一个大版本。每个新版本都会带来易用性、稳定性、安全性、性能等方面的提升，以及一些试验性质的功能。而且版本升级过程也变得越来越简单。那么显而易见，请尽量把你的数据库及时升级到最新的稳定

版。关于每个版本引入的关键特性列表，请参见官方提供的“PostgreSQL 各版本功能特性一览表”（http://www.postgresql.org/about/featurematrix）。

1.4.1 为什么要升级

如果你正在使用 PostgreSQL 8.4 或者更早期的版本，请立即升级！因为 8.4 版在 2014 年 7 月已进入生命周期终结（End of Life，EOL）状态，此后不再提供官方的升级支持。请参考 PostgreSQL 官方的发行版支持策略（http://www.postgresql.org/support/versioning/）以获取更多关于 PostgreSQL EOL 政策的细节。请务必不要使用已过了 EOL 期限的版本，因为开发组不会再为其提供新的安全更新和功能补丁。一旦这种老版本出了问题，你只能花钱去请 PostgreSQL 专家级顾问来解决故障或寻找临时解决方案，这种服务一般都是很昂贵的，而且你不一定能找得到这种专家。

不管当前使用的是哪个大版本，你都应该尽快跟进小版本号的更新。比如从 8.4.17 升级到 8.4.21，只需要替换二进制文件并重启一下即可。小版本仅修改 bug 而不会涉及功能变化，因此这种升级是很安全的，也会为你降低出问题的几率。

1.4.2 PostgreSQL 9.4版中引入的新特性

本书写作期间，PostgreSQL 9.3 是最新的稳定发行版，9.4 还只是测试版，但其安装包已可下载用于测试。9.4 测试版包含以下特性。

- 物化视图特性的改进。在 9.3 版中，刷新物化视图期间会对其加锁并禁止访问。但一般来说刷新物化视图会需要一定的时间，因此在生产环境中该刷新动作会导致物化视图可用性显著降低。9.4 版中取消了刷新时的加锁动作，因此即使是正在被刷新的物化视图也可被访问。但请注意：利用此特性的前提是物化视图必须要拥有一个唯一索引。
- 新增了对 SQL:2008 标准中规定的 `percentile_disc`（不连续百分比）和 `percentile_cont`（连续百分比）这两个分析函数的支持，须配合 `WITHIN GROUP (ORDER BY...)` 子句使用。详细例子可参见 depesz 博客网站的一篇关于 ORDERED SET WITHIN GROUP 聚合运算的介绍文章（http://www.depesz.com/2014/01/11/waiting-for-9-4-support-ordered-set-within-group-aggregates/）。这些函数为你提供了系统原生的快速取中间值功能。例如，如果我们希望从一批考试成绩中取中间点到 3/4 处部分的值，可执行以下查询：

  ```
  SELECT subject, percentile_cont(ARRAY[0.5, 0.75])
    WITHIN GROUP (ORDER BY score) As med_75_score
   FROM test_scores GROUP BY subject;
  ```

  在 PostgreSQL 中，要实现 `percentile_cont` 和 `percentile_disc`，可以取一个数组或 0 到 1 之间的单个值（此值代表所希望查询的百分比范围），并且此实现会相应地返回一个值数组或单个值。语句中的 `ORDER BY score` 表示希望根据 `score` 字段的值来进行百分比计算。

- 创建视图时支持 `WITH CHECK OPTION` 子句，其作用是确保在视图上执行更新或者插入操作时，修改后或者新插入的记录仍然是在本视图可见范围内。详见第 7 章中的示例 7-2。
- 新增对 `jsonb` 数据类型的支持，该数据类型是 JSON（JavaScript Object Notation）类型的二进制存储版本，并且支持索引。通过 `jsonb` 类型可以对 JSON 格式的文档数据建立索引，并可加快对其内部元素的访问速度。详细信息请参考 5.6 节，同时可参考这两篇博客文章："引入新的 jsonb 数据类型：JSON 类型的结构化存储格式"（http://www.depesz.com/2014/03/25/waiting-for-9-4-introduce-jsonb-a-structured-format-for-storing-json/）以及"jsonb：通配符查询"（http://obartunov.livejournal.com/177977.html）。
- GIN 索引的查询速度提升，同时占用空间减少。GIN 索引的使用范围日益广泛，并且非常适用于全文搜索、三连词处理、`hstore` 键值数据库以及 `jsonb` 类型支持等场景。在很多情况下你甚至可以把它当作 B- 树索引的一个替代品，而且一般来说 GIN 索引占用的空间会更少。详情请参见"使用 GIN 索引来代替位图索引"这篇文章的介绍（http://hlinnaka.iki.fi/2014/03/28/gin-as-a-substitute-for-bitmap-indexes）。
- 支持更多 JSON 函数。请参见 Depesz 博客站的文章"9.4 版中的新 JSON 函数介绍"（http://www.depesz.com/2014/01/30/waiting-for-9-4-new-json-functions/）。
- 支持使用以下语法轻松地将所有资产从一个表空间移动到另一个表空间中：`ALTER TABLESPACE old_space MOVE ALL TO new_space;`。
- 支持对返回的结果集中的记录加上数字编号。当我们从数组、`hstore`、复合类型等格式数据源中取出非格式化数据时，由于缺少可用于唯一标识记录的主键，因此一般需要为每条记录加一个数字型的行号。现在可以将系统列 `ordinality`（该列是在 ANSI SQL 标准中定义的）添加到输出。以下是一个使用 `hstore` 对象以及返回一个键值对的 `each` 函数的例子：

```
SELECT ordinality, key, value
 FROM each('breed=>pug,cuteness=>high'::hstore) WITH ordinality;
```

- 支持通过执行 SQL 命令来更改系统配置设置。`ALTER system SET ...` 语法可实现对全局系统设置的动态修改，这一功能在之前版本中只能通过修改 postgresql.conf 文件才能实现。关于 postgresql.conf 文件用法的详情请参见 2.1.1 节。
- 支持对外部表建触发器。通过该功能，即便外部数据源与你相隔万里之遥，只要对方一修改数据，你立即就可以得到通知。不过目前我们尚不确定该功能的实际使用效果到底如何，因为在数据源极其遥远的情况下，由于存在网络延迟，其效果就很难说了。
- 新增 `unnest` 函数，该函数以可预见的方式将不同大小的数组分配到各个列中。
- 新增 `ROWS FROM` 语法，该语法可以将多个函数返回的结果集逐行拼接起来，最后作为一个完整的结果集返回，因此即使这些结果集之间的元素个数不一致也没关系，如下例所示：

```
SELECT * FROM ROWS FROM (
jsonb_each('{"a":"foo1","b":"bar"}'::jsonb),
```

```
jsonb_each('{"c":"foo2"}'::jsonb)) x
(a1,a1_val,a2_val);
```

- 支持使用 C 对动态后台工作线程进行编码以按需完成工作。contrib/worker_spi 目录下的 9.4 版源码中实现了一个小型的示例可供参考。

1.4.3 PostgreSQL 9.3版新特性列表

9.3 版（发布于 2013 年）中引入的主要特性如下。

- 增加了对 ANSI SQL 的标准 `LATERAL` 子句的支持。`LATERAL` 语法允许将 `FROM` 子句与关联一起使用以引用关联的另一方上的变量。在一个关联查询语句中，如果参与关联的一方是一个临时结果集，比如一个子查询或者一个能够返回结果集的函数，那么在该子查询或者函数中可以引用关联关系另一方结果集中的列。如果没有 `LATERAL` 语法，这种横向关联是不可能的，只有 `WHERE` 关联条件部分才能进行被关联方之间列的交叉引用。如果你需要使用 `unnest`、`generate_series` 或者以正则表达式作为查询条件的查询语句来构造关联一方的结果集，那么 `LATERAL` 语法几乎是不可或缺的，因为这种情况下势必要参考关联关系另一方的数据才能生成自己一方的数据。请参考 7.6 节以了解更多细节。
- 支持并行 `pg_dump`。从 8.4 版开始支持并行恢复，在该版本中实现了对并行备份的支持，这可以大大加快大型数据库的备份速度。
- 支持物化视图（详细信息请参见 7.1.3 节）。物化视图可以将经常使用的视图中的数据进行持久化，从而避免反复执行相同的查询。
- 支持自动可更新视图。对视图执行更新操作不再需要创建触发器或者规则，现在可以直接针对视图执行 `UPDATE` 操作，该操作在系统内部会自动映射到此视图的基表上。
- 支持定义基于递归 CTE 表达式的视图。
- 支持更多针对 JSON 类型的构造函数和解析函数。详见 7.6 节。
- 支持在基于正则表达式条件的查询中使用索引。
- 引入了支持 64 位大对象操作的 API，该 API 可操作 TB 级大小的对象。之前最多支持到 2 GB。
- postgres_fdw 驱动程序（10.3.3 节中会进行介绍）支持对其他 PostgreSQL 数据库（即使在使用较低版本 PostgreSQL 的远程服务器上）进行读写操作。伴随此更改的是对实施可写功能的 FDW API 的升级。
- 复制功能做了大量改进，其中最主要的两点是：实现了复制功能的架构无关性，即复制的架构不依赖于特定的操作系统或者硬件；另外支持了基于流式复制的重新选主过程。
- 支持使用 C 语言创建后台工作线程，可用于执行一些定时任务。
- 支持对 DDL 操作定义触发器。
- 支持新的 psql 命令：`watch`。详细信息参见 3.4.2 节。
- 支持新的 `COPY DATA` 命令，可用于 PostgreSQL 与外部程序之间导入或者导出数据。3.5.3 节有详细介绍。

1.4.4 PostgreSQL 9.2版新特性列表

9.2 版（2012 年 9 月发布）中引入的主要特性如下。

- 支持索引内查询。如果查询的列都位于索引内，那么 PostgreSQL 会省略不必要的表访问动作，仅依据索引自身数据就可以完成全部查询动作。该优化会给键值查询以及类似 `COUNT(*)` 这种聚合查询带来巨大的性能提升。
- 内存排序操作性能提升可达 20%。
- 在预处理语句中做了一些改进。现在会对预处理语句进行解析、分析和重写，但你可以跳过该计划来避免被绑定到特定实参输入上。你现在还可以保存依赖实参的某个预处理语句的计划。这样会降低预处理语句比等效的即席查询执行得更差的几率。[4]
- 支持级联流式复制，即支持从一个 slave 节点到另一个 slave 节点的流式复制。
- 支持基于空间分区树算法的 SP-GiST 索引，这种索引是 GiST 索引的强化版本，对于很多依赖 GiST 索引的扩展包来说，使用这种索引可以得到巨大的性能提升。
- 新增了对 `ALTER TABLE IF EXISTS` 语法的支持，修改表结构之前可以不用再检查此表是否存在。
- `ALTER TABLE` 和 `ALTER TYPE` 这两个命令新增了大量功能选项，本来要实现这些功能的话需要删除表并重建。更多细节请参见 depesz 博客网站的“More Alter Table Alter Types”这篇博文（http://www.depesz.com/2012/02/14/waiting-for-9-2-more-rewrite-less-alter-table-alter-types/）。
- 增加了更多的 `pg_dump` 和 `pg_restore` 选项。细节请参见“9.2 版中 `pg_dump` 工具的增强”（http://www.postgresonline.com/journal/archives/252-PostgreSQL-9.2-pg_dump-enhancements.html）。
- 新增了对 PL/V8 语法扩展包的支持，以后可以使用目前非常流行的 JavaScript 语言来编写函数。
- JSON 正式成为系统内置数据类型，同时转正的还有 `row_to_json` 和 `array_to_json` 函数。使用 Ajax 的 Web 开发人员应该会很欢迎该特性。详见 7.6 节和示例 7-16。
- 新增了对区间数据类型的支持，不再需要通过编写函数来实现类似功能。区间类型虽然

注 4：语句的 PREPARE 预解析机制有改进。在之前版本中进行 PREPARE 操作时，规划器会对预解析的语句生成执行计划，但由于此时的语句中不携带具体的选项值（预提交的语法形式如 `PREPARE my_stmt AS select a from tab where b=$1 and c>$2`），因此规划器会基于表的统计信息来进行推测，并生成一个通用的、比较保守的执行计划，这往往会导致得到的执行计划并非最优，从而出现语句经过预解析后再执行反而比直接执行更慢的情况。为解决此问题，当前版本中的 PREPARE 操作有所变化，仍然会执行语句解析、语句分析、语句改写这几个过程，但会跳过生成执行计划这一步骤，留到语句 EXECUTE 阶段根据真正的选项再生成执行计划，这样就保证了执行计划是最优的。但这带来的另一个问题就是 EXECUTE 效率降低，因为每次都要生成执行计划。为此系统做了另一个优化：如果 EXECUTE 阶段生成的执行计划与原来在 PREPARE 阶段生成的通用执行计划效率类似，那么系统还是会将通用执行计划存储下来并复用。通过以上策略，避免了有时执行预提交的语句反而更慢的情况。——译者注

是首次引入，但系统为其提供了丰富的运算符和配套函数。Exclusion 类型的约束可以很好地保证该类型数据的合法性。
- 支持在 SQL 函数中通过实参名称引用实参，之前版本仅支持通过实参编号引用实参。有多个实参时，通过实参名称引用会更方便。

1.4.5 PostgreSQL 9.1版新特性列表

在 9.1 版中，PostgreSQL 推出了诸多企业级特性，开始与 SQL Server 和 Oracle 等重量级对手展开正面竞争。

- 内置的复制特性全面增强，支持同步复制。
- 引入新的 `CREATE EXTENSION` 和 `ALTER EXTENSION` 命令来管理扩展包，安装和卸载扩展包变得轻而易举。
- 引入兼容 ANSI 标准的外部数据封装器（FDW）机制，用于查询外部数据源。
- 支持可写 CTE 表达式，此后 `UPDATE` 和 `INSERT` 语句也可以享受到 CTE 语法带来的便利。
- 支持建立无日志的表，以加快写操作速度。有的场景下，记录预写日志确无必要。
- 支持针对视图建立触发器。在之前的版本中，如果希望对视图进行修改操作，只能通过建立 `DO INSTEAD` 类型的规则（详情请参见官网链接：http://www.postgresql.org/docs/current/interactive/rules-update.html）来实现，且这种规则只能用 SQL 语言编写，但支持视图触发器后，就可以用上 PostgreSQL 支持的所有语言，这样就可以针对视图实现更复杂更抽象的业务逻辑。
- 引入 KNN GiST 索引类型，该类索引能对常用的很多扩展包带来性能提升，比如全文检索、三连词（用于模糊查找以及区分大小写查找操作）以及 PostGIS 等。

1.5 数据库驱动程序

任何情况下，你都不可能脱离具体的业务系统而仅仅使用 PostgreSQL 数据库本身，那显然是无意义的。为了实现 PostgreSQL 与业务系统之间的交互，就需要借助数据库驱动程序。PostgreSQL 拥有大量免费驱动，支持各种编程语言和开发工具。此外，很多商业公司也以很低廉的价格提供了各有特色的驱动。目前比较流行的几种开源驱动如下。

- PHP 驱动：PHP 语言广泛应用于 Web 开发领域，大多数 PHP 发行包都自带了较老的 pgsql 驱动或者是较新的 `pdo_pgsql` 驱动。一般来说这两种驱动默认都会安装，不过可能需要修改 php.ini 来决定启用哪一种。
- JDBC 驱动：JAVA 开发所使用的 JDBC 驱动一直是与最新版 PostgreSQL 同步更新的，可以从 PostgreSQL 官方站点下载（https://jdbc.postgresql.org）。
- .NET 驱动：.NET 框架（含微软的官方版和 Mono 社区的开源版）可使用 Npgsql 驱动（http://npgsql.projects.pgfoundry.org）。目前该驱动支持微软 .NET 框架 V3.5 及之后的版本，包括微软 Entity Framework 开发框架以及 Mono 开源 .NET 框架。

- ODBC 驱动：如果需要从微软 Access、Office 系列工具或者其他支持 ODBC 的产品连接到 PostgreSQL，可从 PostgreSQL 官网下载 ODBC 驱动（http://www.postgresql.org/ftp/odbc/versions/msi），支持 32 位和 64 位两个版本。
- LibreOffice/OpenOffice 驱动：LibreOffice 3.5 及之后的版本中自带了 PostgreSQL 驱动，但 3.5 之前的版本以及 OpenOffice 是不带的，可以使用 JDBC 或者 SDBC 驱动。更多细节请参见“OO Base 与 PostgreSQL”这篇博文（http://www.postgresonline.com/journal/categories/23-oobase）。
- Python 驱动：Python 可通过多种驱动（https://wiki.postgresql.org/wiki/Python）访问 PostgreSQL，目前 psycopg（http://initd.org/psycopg/）是最流行的一种。Python 的 Django 开发框架（https://docs.djangoproject.com/en/dev/ref/databases/#postgresql-notes）对 PostgreSQL 也有着良好的支持。
- Ruby 驱动：对 Ruby 开发人员来说，请使用 rubygems pg 驱动（https://rubygems.org/gems/pg）。
- Perl 驱动：Perl 可以使用 DBI 和 DBD::Pg 驱动。也可以使用由 CPAN 网站（http://search.cpan.org/modlist/Database_Interfaces）提供的 DBD::PgPP 驱动。
- Node.js 驱动：Node.js 是一个基于 Google V8 JavaScript 引擎的运行平台，可用于构建高可扩展性的网络应用。该平台目前支持三种 PostgreSQL 驱动：Node Postgres（https://github.com/brianc/node-postgres）、Node Postgres Pure（与前者的区别在于不需要编译，https://github.com/brianc/node-postgres-pure），以及 Node-DBI（https://github.com/DrBenton/Node-DBI）。

1.6 如何获得帮助

在使用 PostgreSQL 过程中，你迟早会需要寻求帮助，我们希望你能够尽早了解到求助的途径。我们最为推荐的途径是邮件列表，不管你是 PostgreSQL 的新用户还是老用户，邮件列表都能为你解答技术问题。可以先打开 PostgreSQL 邮件列表页面（http://www.postgresql.org/community/lists/），该页面上有各种邮件列表的信息以及订阅方法。如果你是新手，那么订阅 PGSQL-General 这个邮件列表（该邮件列表的历史信息归档地址：http://archives.postgresql.org/pgsql-general）是最合适的。如果你发现了 PostgreSQL 的 bug，那么打开“PostgreSQL 故障报告”这个页面（http://www.postgresql.org/docs/current/interactive/bug-reporting.html），上面会告诉你具体如何操作。

1.7 PostgreSQL的主要衍生版本

PostgreSQL 使用了 MIT/BSD 风格的许可证，任何人都可以合法地对其修改并二次传播，因此对于那些想创建自己数据库分支的人来说，PostgreSQL 是绝佳的选择。在过去的很多年间，有很多团队创建了自己的 PostgreSQL 衍生版本，其中的部分修改也已经回馈到

PostgreSQL 的主干代码中。

目前数据仓库领域使用很广泛的 Netezza（http://www.netezza.com）就是源自 PostgreSQL。亚马逊公司的 Redshift 数据仓库（http://aws.amazon.com/redshift/）事实上是 PostgreSQL 的一个分支的分支。支持 PB 级数据分析的著名数据仓库 GreenPlum 最初的源头是 Bizgres，而 Bizgres 是一款基于 PostgreSQL 的面向大数据的数据仓库和智能分析软件。EnterpriseDB 公司（http://enterprisedb.com）推出的 PostgreSQL Advanced Plus 也是以 PostgreSQL 为基础，另外增加了对于 Oracle 语法和特性的兼容支持，以吸引原 Oracle 用户。EnterpriseDB 公司向 PostgreSQL 社区提供了资金和开发力量的支持，对此我们表示感谢。他们的 Postgres Plus Advanced Server 产品在版本更新节奏上也一直是密切跟进最新的 PostgreSQL 稳定版的。

前述衍生产品都是商业化的闭源软件。tPostgres（http://www.tpostgres.org）、Postgres-XC（http://postgres-xc.sourceforge.net）和 Big SQL（http://www.bigsql.org）是三款还处于发展初期但已经崭露头角的开源衍生产品，它们都得到了 OpenSCG 公司（http://www.openscg.com/）的资金支持。tPostgreSQL 的最新版本基于 PostgreSQL 9.3，其目标是取代 Microsoft SQL Server。tPostgreSQL 中内嵌了 pgtsql 语言扩展包，可以用 T-SQL 语法来编写函数。pgtsql 语言包是标准的 PostgreSQL 扩展包，因此它其实可以安装到任何一台 PostgreSQL 9.3 数据库上。Postgres-XC 是一套集群服务器系统，它能够提供可扩展的写能力并支持同步多主复制，其分布式处理和多主复制能力使它在所有类似系统中脱颖而出。目前 Postgres-XC 还只是 1.0 版本。最后介绍一下 BigSQL，它实现了 PostgreSQL 和 Hadoop with Hive 这两款重量级产品的融合。BigSQL 自带了 `hadoop_fdw` 这款扩展包，可以查询和更新外部 Hadoop 数据源的数据。

此外，最近还发布了一款 PostgreSQL 开源分支产品 Postgres-XL（XL 代表 eXtensible Lattice，即可扩展的晶格。http://www.postgres-xl.org/），该产品面向大规模并行处理（MPP）领域，支持节点间数据的分片存储能力。

第2章
数据库管理

本章涵盖了管理 PostgreSQL 最常用的一些基本操作，包括角色与权限管理、数据库的创建、插件安装、数据备份与恢复等。我们假定你已经安装好了一套 PostgreSQL 及其相应的管理工具，并且你有权任意地调整和使用该套环境，切记勿在生产环境上执行测试动作。

2.1 配置文件

配置文件控制着一个 PostgreSQL 服务器实例的基本行为，主要包含以下几个文件。

- postgresql.conf

  该文件包含一些通用设置，比如内存分配、新建 database 的默认存储位置、PostgreSQL 服务器的 IP 地址、日志的位置以及许多其他设置。9.4 版中引入了一个新的 postgresql.auto.conf 文件，任何时候执行 `ALTER SYSTEM` SQL 命令，都会创建或重写该文件。该文件中的设置会替代 postgresql.conf 文件中的设置。

- pg_hba.conf

  该文件用于控制访问安全性，管理客户端对 PostgreSQL 服务器的访问权限，内容包括：允许哪些用户连接到哪个数据库，允许哪些 IP 或者哪个网段的 IP 连接到本服务器，以及指定连接时使用的身份验证模式。

- pg_ident.conf

  pg_hba.conf 的权限控制信息中的身份验证模式字段如果指定为 ident 方式，则用户连接时系统会尝试访问 pg_ident 文件，如果该文件存在，则系统会基于文件内容将当前

执行登录操作的操作系统用户映射为一个 PostgreSQL 数据库内部用户的身份来登录。有些人会把操作系统的 root 用户映射为 PostgreSQL 的 `postgres` 超级用户账号。pg_hba.conf 中的每条权限控制信息均可以指定一个独立的 pg_ident.conf 文件作为用户映射信息数据源。

如果你在安装过程中使用了默认配置，则上述文件会位于 PostgreSQL 主数据文件夹中。你可以使用任何文本编辑器来编辑这些文件，或者在 pgAdmin 工具中专门的功能页面上也可以进行修改（在 pgAdmin 工具中直接修改系统配置文件的功能是基于 PostgreSQL 提供的 Admin Pack 系列管理 API 实现的）。4.2.2 节中会有关于如何在 pgAdmin 工具中修改 postgresql.conf 和 pg_hba.conf 文件的介绍。如果你不确定这些文件的具体位置，请以超级用户身份连接到任何一个数据库上并执行示例 2-1 中的查询语句即可找到。

示例 2-1：配置文件的位置

```
SELECT name, setting FROM pg_settings WHERE category = 'File Locations';

       name        |                 setting
-------------------+-----------------------------------------
 config_file       | /etc/postgresql/9.3/main/postgresql.conf
 data_directory    | /var/lib/postgresql/9.3/main
 external_pid_file | /var/run/postgresql/9.3-main.pid
 hba_file          | /etc/postgresql/9.3/main/pg_hba.conf
 ident_file        | /etc/postgresql/9.3/main/pg_ident.conf
```

2.1.1 postgresql.conf

postgresql.conf 文件包含了 PostgreSQL 服务能够正常运行所必需的基础设置以及新建数据库时所使用的默认设置。你可以在数据库级、用户级、会话级甚至是函数级替代这些设置。“对你的 PostgreSQL 服务器进行调优”（https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server）这篇文章详细介绍了如何通过修改设置来对你的 PostgreSQL 系统进行优化。

通过查询 `pg_settings` 视图可以很方便地检查当前设置，示例 2-2 展示了具体语法。该示例中查询了主要的几个关键设置并给出了查询结果中每一列的含义说明，但如果你希望更深入地了解这些设置，我们建议你参考官方手册中的相关介绍（http://www.postgresql.org/docs/current/interactive/view-pg-settings.html）。

示例 2-2：关键的设置

```
SELECT name, context ❶, unit ❷,
    setting, boot_val, reset_val ❸
FROM pg_settings
WHERE name IN ( 'listen_addresses', 'max_connections', 'shared_buffers', 'effec
tive_cache_size', 'work_mem', 'maintenance_work_mem'
)
ORDER BY context, name;
```

```
         name          | context    | unit | setting | boot_val  | reset_val
-----------------------+------------+------+---------+-----------+-----------
 listen_addresses      | postmaster |      | *       | localhost | *
 max_connections       | postmaster |      | 100     | 100       | 100
 shared_buffers        | postmaster | 8kB  | 131584  | 1024      | 131584
 effective_cache_size  | user       | 8kB  | 16384   | 16384     | 16384
 maintenance_work_mem  | user       | kB   | 16384   | 16384     | 16384
 work_mem              | user       | kB   | 5120    | 1024      | 5120
```

❶ 如果将 `context` 设置为 `postmaster`，那么更改此形参后需要重启 PostgreSQL 服务才能生效；如果将其设置为 `user`，那么只需要执行一次重新加载即可全局生效。重启数据库服务会终止活动连接，但重新加载不会。

❷ `unit` 字段表示这些设置的单位。示例 2-2 的输出结果中内存设置的单位可能会让你觉得有点乱，有些是以 8 KB 为单位，有些是以 KB 为单位。在 postgresql.conf 中设置内存时，请尽量选择一个合适的数值单位，比如我们要将内存设定为 128 MB，那么“xxx = 128 MB”就是一个比较好的写法，而“xxx = 131072 KB”显然就不太好，虽然二者是等价的。你也可以通过 SHOW 命令查看特定系统设置，比如：`SHOW effective_cache_size` 或 `SHOW maintenance_work_mem`。`SHOW` 命令的输出结果会自动根据数值大小选择合适的单位。如果希望一次性查看使用了合适单位的所有设置，可以使用 `SHOW ALL` 命令，其输出结果会针对每个设置选用合适的单位。

❸ `setting` 是指当前设置；`boot_val` 是指默认设置；`reset_val` 是指重新启动服务器或重新加载设置之后的新设置。在 postgresql.conf 中修改了设置后，一定要记得查看一下 `setting` 和 `reset_val` 并确保二者是一致的，否则说明设置并未生效，需要重新启动服务器或者重新加载设置。

请特别注意 postgresql.conf 中的以下网络设置，修改这些值是一定要重新启动数据库服务的。



对于 9.4 版及之后的版本来说，postgresql.auto.conf 的优先级是高于 postgresql.conf 的，如果这两个文件中存在同名配置项，则系统会优先使用前者设定的值。

- `listen_addresses`

  表示 PostgreSQL 服务使用的 IP 地址，一般会设定为 `localhost` 或者 `local`，但也有很多人会设为 `*`，表示使用本机任一 IP 地址均可连接到 PostgreSQL 服务。

- `port`

  PostgreSQL 服务的侦听端口，默认值为 5432。如果是在 Red Hat 或者 CentOS 平台上，可以更改 `PGPORT` 值 /etc/sysconfig/pgsql/your_service_name_here 来更改侦听端口。

- `max_connections`

  系统允许的最大并发连接数。

按照我们的经验，以下四个设置对系统性能有着全局性的影响，我们建议你在实际环境下通过实测来找到最优值。

- `shared_buffers`

  此设置定义了用于缓存最近访问过的数据页的内存区大小，所有用户会话均可共享此缓存区。此设置对查询速度有着重大影响，一般来说是越大越好，至少应该达到系统总内存的 25%，但不宜超过 8 GB，因为超过后会出现“边际收益递减”效应，即消耗的内存很多，但得到的速度提升却很少，得不偿失。修改此设置需要重启 PostgreSQL 服务。

- `effective_cache_size`

  此设置表示一个查询执行过程中可以使用的最大缓存，包括操作系统使用的部分以及 PostgreSQL 使用的部分。系统并不会根据这个值来真实地分配这么多内存，但是规划器会根据这个值来判断系统能否提供查询执行过程中所需的内存。如果将此设置设得过小，远远小于系统真实可用内存量，那么可能会给规划器造成误导，让规划器认为系统可用内存有限，从而选择不使用索引而是走全表扫描（因为使用索引虽然速度快，但需要占用更多的中间内存）。在一台专用于运行 PostgreSQL 数据库服务的服务器上，建议将 `effective_cache_size` 的值设为系统总内存的一半或者更多。此设置的更改可动态生效，执行重新加载即可。

- `work_mem`

  此设置指定了用于执行排序、哈希关联、表扫描等操作的最大内存量。要得到此设置的最优值需要考虑以下一些因素：数据库的使用方式，需要预留多少内存给除数据库系统外的程序，以及服务器是否专用于运行 PostgreSQL 服务等问题。如果使用场景仅仅是有很多用户并发执行简单查询，那么这个值设得很小也没问题。关于 `work_mem` 设置有一篇很好的文章“理解 `work_mem`”（http://www.depesz.com/2011/07/03/understanding-postgresql-conf-work_mem/）。此设置的更改可动态生效，执行重新加载即可。

- `maintenance_work_mem`

  此设置指定可用于 vaccum 操作（即清空已标记为“被删除”状态的记录）这类系统内部维护操作的内存总量。其值不应大于 1 GB。此设置的更改可动态生效，执行重新加载即可。

上述设置可在库级、用户级以及函数级设置。例如，如果有一个精通 SQL 的用户要在库上执行非常复杂的 SQL 语句，那么可以为此用户单独调大 `work_mem` 的值。又比如有一个函数中有很多排序操作，那么可以仅针对此函数调大 `work_mem` 的值。

PostgreSQL 9.4 版中引入了对新的 `ALTER SYSTEM` SQL 命令的支持，使用该命令可以更改设

置。例如，如果要全局设置 work_mem，执行以下命令即可：

```
ALTER SYSTEM set work_mem = 8192;
```

每个设置有着各自不同的特性，有的更改后必须重启数据库服务才能生效，有的只要重新加载一次就可以了，下面这个命令可以实现设置重新加载：

```
SELECT pg_reload_conf();
```

PostgreSQL 记录更改是在一个称为 postgresql.auto.conf 的替代文件中通过 ALTER SYSTEM 所做出的，而不是直接对 postgresql.conf 进行更改。

“遇到修改了postgredql.conf文件，结果服务器崩溃了这种情况。”

定位这种问题最简单的方法是查看日志文件，该文件位于 PostgreSQL 数据文件夹的根目录或者 pg_log 子文件夹下。只要找到最近修改的那个日志文件并查看其最后部分的内容就能找到本次问题的相关错误日志，日志的内容一般都是比较直白易懂的，你看了就会明白。

最常见的错误是把 shared_buffers 设得太大了，还有一个常见问题是由于上次系统异常关闭导致遗留了一个没来得及删除的 postmaster.pid 文件，该文件就位于数据文件夹下，你可以手动删除该文件并重新启动 PostgreSQL。

2.1.2 pg_hba.conf

pg_hba.conf 文件指定了允许哪些用户以何种方式连接到 PostgreSQL 数据库。针对该文件的修改可动态生效。一个典型的 pg_hba.conf 文件看起来如示例 2-3 所示。

示例 2-3：pg_hba.conf 文件示例

```
# TYPE DATABASE USER ADDRESS METHOD
# IPv4 local connections:
host all  all  127.0.0.1/32 ident ❶
# IPv6 local connections:
host all  all ::1/128 ❷trust
host all  all 192.168.54.0/24 ❸md5
hostssl ❹ all all 0.0.0.0/0 md5
# Allow replication connections from localhost, by a user with the ❺
# replication privilege.
#host replication postgres 127.0.0.1/32 trust
#host replication postgres ::1/128 trust
```

❶ 身份验证模式。一般有以下几种常用选项：ident、trust、md5 以及 password。从 9.1 版开始引入了 peer 身份验证模式（详情请参考官方手册相关章节：http://www.postgresql.org/docs/current/interactive/auth-methods.html）。ident 和 peer 模式仅适用于 Linux、Unix 和 Mac，不适用于 Windows。一些比较少见的身份验证模式，比如 gss、radius、ldap 以及 pam 等，可能并不会在所有的发行版中都默认安装。

❷ 用于定义网络范围的 IPv6 语法。只有服务器支持 IPv6 时才可以配置该项，如果在非 IPv6 网络环境下配置了这样的条目，系统会直接忽略该配置项。

❸ 用于定义网络范围的 IPv4 语法。第一部分（本例中为 `192.168.54.0`）是网络地址，后面跟着的 `/24` 是位掩码。本例中这样定义可以达到如下效果：对于任何一个位于 192.168.54.0 子网中的客户端来说，只要该客户端提供的经 md5 算法加密的密码是正确的，那么系统就允许该客户端连到数据库服务器。

❹ 这是针对 SSL 连接的规则。本例中，任何使用 SSL 连接并能提供合法 md5 加密密码的客户端都可以连接到本地数据库服务器。

❺ 此处是一套复制系统中其他成员节点的 IP 地址列表，每个成员的地址都必须存在于此列表中，否则无法加入该复制系统。该特性从 9.0 版开始引入。本例中，这几行是注释掉的。

对于每一个连接请求，`postgres` 服务会按照 pg_hba.conf 文件中记录的规则条目自上而下进行检查。当匹配到第一条允许此请求接入的规则时，就不再往下检查，系统将允许该连接请求。类似地，如果匹配到一条拒绝此连接请求的规则，也不再继续检查，并拒绝该连接请求。如果一直搜索到文件的末尾都没能找到匹配项，那么按照默认规则处理，即拒绝该连接。大家常犯的一个错误是把规则的顺序放错。例如，如果你将 `+0.0.0.0/0 reject+` 规则放到 `+127.0.0.1/32 trust+` 的前面，那么此时本地用户全都无法连接，即使下面有规则允许也不行。

1. “遇到修改了ph_hba.conf文件，结果服务器崩溃了这种情况。”

不用担心，这种事情经常发生，解决起来不难。这一般是因为拼写错误或增加了一种不支持的身份验证模式导致。如果 `postgres` 服务无法正确地解析 pg_hba.conf 文件，那么为确保系统安全它会禁止所有的连接请求甚至是禁止系统启动。最简单的诊断方法是看一下日志，文件就在数据文件夹的根目录下或者其 pg_log 子文件夹下。可以打开修改日期最近的日志文件并看一下最后部分的内容，错误提示信息一般就在那里，而且一般都是很好理解的。如果你经常会笔误改错东西，那么请一定记得在修改配置文件之前做个备份。

2. 身份验证方法

PostgreSQL 提供了多种模式用于用户身份验证，很可能是所有数据库里面支持的模式最多的。大多数人只会用到其中几种最常见的：`trust`、`peer`、`ident`、`md5` 和 `password`。还有一种 `reject` 模式，其作用是拒绝所有请求。pg_hba.conf 中定义的这些身份验证规则就好像是整个 PostgreSQL 服务器的看门人，确保着整个系统的外部访问安全。当然，在通过了这一层外部安全控制并成功建立连接以后，连上来的用户仍需遵守角色权限和数据库访问限制等内部约束规则。

如需了解有关各种身份验证方法的详细内容，请参考官方手册中的“PostgreSQL 客户端身份验证”（http://www.postgresql.org/docs/current/interactive/client-authentication.html）。最常

用的身份验证方法有以下这些。

- `trust`

  这是最不安全的身份验证模式。该模式允许用户“自证清白”，即可以不用密码就连接到数据库。只要源端 IP 地址、连接用户名、要访问的 database 名都与该条规则匹配，用户就可以连上来。`trust` 模式很不安全，因此应对其使用予以限制，即只能允许从数据库服务器本机发起的连接或者是同属内网的用户发起的连接使用此模式。但即使加了前述限制也不能保证安全，因为会有人通过伪装 IP 地址的方式来冒用此权限，所以有人认为该模式应该被彻底禁用。然而在单用户的桌面环境下这却是最常用的身份验证模式，因为一般这种场景下系统的安全性根本不是问题。连接时如果未指定用户名，那么默认会使用当前登录的操作系统用户名。

- `md5`

  该模式很常用，要求连接发起者携带用 md5 算法加密的密码。

- `password`

  不推荐，因为该模式使用明文密码进行身份验证，不安全。

- `ident`

  该身份验证模式下，系统会将请求发起者的操作系统用户映射为 PostgreSQL 数据库内部用户，并以该内部用户的权限登录，且此时无需提供登录密码。操作系统用户与数据库内部用户之间的映射关系会记录在 pg_ident.conf 文件中。

- `peer`

  该模式使用连接发起端的操作系统名进行身份验证。仅可用于 Linux、BSD、Mac OS X 和 Solaris，并且仅可用于本地服务器发起的连接。

多种身份验证模式是可以同时使用的，即使是针对同一个 database 也可以这么做，也就是说我们可以针对同一个 database 设置多条身份验证规则，并且每条规则的身份验证模式都不一样。虽然身份验证模式很灵活，但请你务必牢记 PostgreSQL 对于 pg_hba.conf 中的规则的查找顺序是从上到下，第一条匹配到的规则就是系统使用的规则。

2.1.3 配置文件的重新加载

很多（但非全部）配置文件更改后必须要重启 `postgres` 服务才能生效，但另外一些只需要执行一次重新加载即可生效。重新加载的过程并不会中断当前已建立的连接。打开一个命令行窗口执行以下命令即可。

```
pg_ctl reload -D your_data_directory_here
```

如果你在 RedHat Enterprise Linux、CentOS 或者 Ubuntu 上是以服务的形式安装的

PostgreSQL，那么请执行以下命令。

```
service postgresql-9.3 reload
```

上述命令中的 `postgresql-9.3` 是服务名。对于较老的版本来说，服务名可能就叫 `postgresql`，不带版本号。

另一种重新加载配置文件的方法是以超级用户权限登录到任何一个 database 并执行以下 SQL 语句：

```
SELECT pg_reload_conf();
```

另外也可以从 pgAdmin 工具中执行重新加载，请参考 4.2.2 节。

2.2　连接管理

我们可能时不时地会遇到一些想要终止数据库连接的情况，比如有人执行了写得很糟糕的 SQL 语句把系统资源耗光，当然这肯定不是他的本意，又比如你在执行某些语句时发现其耗时太长，超出了自己忍耐的极限。发生这些情况时，我们一般都会希望结束这些操作或者干脆彻底终止这个连接。另外，当我们执行全库备份、全库恢复或者对有人正在访问的表执行数据恢复时，我们都会需要先终止一些相关连接。下面将介绍具体的操作过程。

请记住，强行终止连接是一种很不“优雅”的行为，应当尽量少用。应当先在客户端应用程序中通过某种方式判定并记录下那些已经失控（耗时长或者占资源多）的语句，然后基于这些记录下来的信息分析出应该终止哪些相关连接。出于礼貌，你应该在终止连接之前通知相关用户其连接即将被强行终止，或者如果实在有必要，你也可以不管它什么礼貌不礼貌，等四下无人时直接终止这些连接就好了。

我们一般会使用以下三个 SQL 语句来取消正在运行的查询并终止连接。以下是典型的流程。

(1) 查出活动连接列表及其进程 ID。

```
SELECT * FROM pg_stat_activity;
```

该命令还能查出每个连接上最近一次执行的语句、使用的用户名（`usename` 字段）、所在的 database 名（`datname` 字段）以及语句开始执行的时间。通过查询该视图可以找到需要终止的会话所对应的进程 ID。

(2) 取消连接上的活动查询。

```
SELECT pg_cancel_backend(procid);
```

该操作不会终止连接本身。

(3) 终止该连接。

```
SELECT pg_terminate_backend(procid);
```

如果你未停止某个连接上正在执行的语句就直接终止该连接，那么这些语句此时也会被停止掉。在上述步骤 2 执行完毕后，客户端应用的挂起状态被解除，即客户端可以重新执行语句，有些着急的用户会在此时再次执行刚刚被终止掉的语句，这又会导致系统陷入之前的状态。为了避免此种情况的发生，可以采用直接终止连接的方式。

PostgreSQL 支持在 `SELECT` 查询语句中调用函数。因此，尽管 `pg_terminate_backend` 和 `pg_cancel_backend` 一次仅能处理一个连接，但你可以通过在 `SELECT` 语句中调用函数的方式实现一次处理多个连接。例如，如果你希望一次性终止某个用户的所有连接，那么在 9.2 版及之后的版本上可以执行以下语句。

```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE usename =
'some_role';
```

在 9.2 版之前的版本上可以执行以下语句。

```
SELECT pg_terminate_backend(procpid) FROM pg_stat_activity WHERE usename =
'some_role';
```

从 9.1 版开始，`pg_stat_activity` 视图发生了较大变化，一些字段的名称发生了变化，并且另外新增了一些字段。原来的 `procpid` 现在叫 `pid`。

2.3 角色

PostgreSQL 中使用“角色”（role）这个术语来表示用户账户的概念。拥有登录数据库权限的角色称为*可登录角色*（login role）。一个角色可以继承其他角色的权限从而成为其*成员角色*（member role）；一个拥有成员角色的角色被称为*组角色*（group role）。设计“组角色”这一功能的本意是为了将一组权限集中在一起成为一个“组”，然后便于以“组”为单位对这些权限进行管理，比如可以通过角色权限继承的方式一次性将这一组权限赋予其成员角色（你可能在想，一个组角色能否是另一个组角色的成员角色？没错，这是可以的，并且这种角色间继承关系可以有无限多层，但除非你非常有把握能搞定这种多层嵌套关系，否则别这么干，因为你最后一定会把自己搞糊涂）。一个拥有登录权限的组角色被称为*可登录的组角色*。然而，为了可维护性和安全性，数据库管理员一般不会为组角色授予登录权限，因为设计组角色的本意是将其作为一个“权限集合”使用，而不是将其作为一个真正需要登录权限的用户角色来使用。一个角色可被授予超级用户（SUPERUSER）权限，拥有此权限的角色可以对 PostgreSQL 进行全面控制。



PostgreSQL 从最近的几个版本开始不再使用“用户”和“组”这两个术语。但在社区讨论版块上你还会看到有人使用这两个术语，请记住“用户”和“组”分别代表“可登录角色”和“组角色”就好了。为保持前向兼容，`CREATE USER` 和 `CREATE GROUP` 这两个命令在当前版本中也是支持的，但我们建议最好不要使用它们，请使用 `CREATE ROLE`。

2.3.1 创建可登录角色

在 PostgreSQL 安装过程中的数据初始化阶段，系统会默认创建一个名为 `postgres` 的角色（同时会创建一个名为 `postgres` 的同名 database）。你可以通过本书前面曾介绍过的 ident 身份验证机制来将操作系统的 root 用户映射到数据库的 `postgres` 角色，这样可以实现 root 用户无密码直接登录。数据库安装完成后，第一件要做的事就是用 psql 或者 pgAdmin 工具以 `postgres` 角色身份登录，然后创建其他已规划好的角色。pgAdmin 工具中有专门的图形界面用于创建角色，如果你希望用 SQL 语句手动创建，请参考示例 2-4 中的 SQL 语句。

示例 2-4：创建具备登录权限的角色

```
CREATE ROLE leo LOGIN PASSWORD 'king' CREATEDB VALID UNTIL 'infinity';
```

`VALID` 行是可选的，其功能是为此角色的权限设定有效期，过期后所有权限都将失效，默认时限是 `infinity`，即永不过期。`CREATEDB` 修饰符表明为此角色赋予了创建新数据库的权限。

如果要创建一个具备超级用户权限的角色，可以参考示例 2-5。当然，要想创建一个超级用户，创建者自身也必须是一个超级用户。

示例 2-5：创建具备超级用户权限的角色

```
CREATE ROLE regina LOGIN PASSWORD 'queen' SUPERUSER VALID UNTIL '2020-1-1 00:00';
```

上面的语句中，我们创建了一个拥有至高无上权力的超级用户“queen”，但我们又不希望这位“queen”永远“统治”下去，那么怎么办呢？用 `VALID` 子句给她的权力加一个期限就好了。

2.3.2 创建组角色

一般不应授予组角色登录权限，因为其作用是将一组权限汇聚成一个集合以便于将这组权限批量授予别的普通角色。当然，这只是我们基于实践经验给出的建议，你也可以为组角色授予登录权限，这完全没问题。

可以用以下 SQL 创建组角色。

```
CREATE ROLE royalty INHERIT;
```

请注意术语 INHERIT 的用法。它表示组角色 royalty 的任何一个成员角色都将自动继承其除“超级用户权限”外的所有权限。出于安全考虑，PostgreSQL 不允许超级用户权限通过继承的方式传递。

以下语句可以将组角色的权限授予其成员角色。

```
GRANT royalty TO leo;
GRANT royalty TO regina;
```

从组角色继承权限

PostgreSQL 有一个很“奇葩”（或者从另一个角度看也可以称之为“方便”）的功能，就是禁止组角色将其权限授予其成员角色，该功能通过 NOINHERIT 关键字控制。因此，创建组角色时请务必显式指明 INHERIT 或者 NOINHERIT 关键字，如果不指明就只能依靠系统默认的设定了，而这个默认设定你必须自己清楚地记住，如果记反了必定会引发问题，为了避免这种事情的发生，我们建议你还是显式指明一下。

有些权限是无法被继承的，例如前面提到过的 SUPERUSER 超级用户权限就无法被继承；然而成员角色可以通过 SET ROLE 命令来实现“冒名顶替”其父角色[1]的身份，从而获得其父角色所拥有的 SUPERUSER 权限，当然这种冒名顶替的状态是有期限的，仅限于当前会话存续期间有效。例如，royalty 组角色的成员角色可以通过执行以下语句来实现上述“冒名顶替”的目的。

```
SET ROLE royalty;
```

请记住这种方法仅适用于会话存续期间，它不是一种永久授权行为，也就是说一旦会话中断，成员角色的 SUPERUSER 权限就会被收回。如果希望将 SUPERUSER 权限永久授予某些成员角色，只能对他们一个一个手动授权。设计这套看似麻烦的机制是为了避免由于误操作而导致 SUPERUSER 权限被错误地授予某个组角色的所有成员角色，而这种情况是极度危险的。

有一个比 SET ROLE some_role 更强大的命令：SET SESSION AUTHORIZATION some_role。这两条命令的主要差别如下所示。

- 首先，只有具备 SUPERUSER 权限的用户才可以执行 SET SESSION AUTHORIZATION，而 SET ROLE 是任何一个成员角色都可以执行的。其次，SET SESSION AUTHORIZATION 能够使当前角色“扮演”系统中任何一个其他角色，即当前角色可以拥有任何其他目标角色的身份与相应权限，而不像 SET ROLE 那样仅仅限于“扮演”其父角色。

注 1：此处首次提到了“父角色”，其实就是“组角色”，此语境下用“父角色”更自然，原文亦使用了 parent role 这个称呼。——译者注

- 从系统内部实现机理上看，每个会话会有两个表示当前用户身份的环境变量：一个是 session_user，即当前用户登录时带的原始身份；一个是 current_user，即当前用户所扮演的身份，默认情况下二者是一致的。SET SESSION AUTHORIZATION 命令会将 current_user 和 session_user 都替换为所“扮演”角色的相应身份 ID，而 SET ROLE 命令只会修改 current_user，而保持 session_user 不变。这意味着 SET SESSION AUTHORIZATION 命令会对后续的 SET ROLE 命令产生影响，因为原始身份 session_user 也发生了变化；而 SET ROLE 命令不会对后续的 SET ROLE 命令产生影响，因为原始身份 session_user 未发生变化。
- 假设某会话的原始身份是 ROLE_A，即 current_user 和 session_user 都是 ROLE_A，然后成功地执行了 SET SESSION AUTHORIZATION ROLE_B 命令，那么 current_user 和 session_user 标识都被修改成了 ROLE_B，之后如果在此会话上再执行 SET ROLE 命令的话，基础身份就是 ROLE_B 了，也就是说此时 SET ROLE 只能设定为 ROLE_B 所归属的某个组角色。但由于 SET ROLE 并不修改 session_user 标识，因此在执行过 SET ROLE 之后再执行 SET ROLE 的话，后一个 SET ROLE 操作的基础身份是不变的，还是当前的 session_user 角色。

2.4　创建database

最基本的创建数据库的 SQL 语句是：

```
CREATE DATABASE mydb;
```

该命令会以 template1 库为模板生成一份副本并将此副本作为新 database，每个 database 都会有一个属主，这个新库的属主就是执行此 SQL 命令的角色。任何一个拥有 CREATEDB 权限的角色都能够创建新的 database。

2.4.1　模板数据库

顾名思义，模板数据库就是创建新 database 时所依赖的模板。创建新 database 时，PostgreSQL 会基于模板数据库制作一份副本，其中会包含所有的数据库设置和数据文件。

PostgreSQL 安装好以后默认附带两个模板数据库：template0 和 template1。如果创建新库时未指定使用哪个模板，那么系统默认会使用 template1 库作为新库的模板。



切记，任何时候都不要对 template0 模板数据库做任何修改，因为这是原始的干净模板，如果其他模板数据库被搞坏了，基于这个数据库做一个副本就可以了。如果你希望定制自己的模板数据库，那么请基于 template1 进行修改，或者自己另外创建一个模板数据库再修改。对基于 template1 或你自建的模板数据库创建出来的数据库来说，你不能修改其字符集编码和排序规则。如果你希望这么干，那么请基于 template0 模板来创建新数据库。

基于某个模板来创建新数据库的基本语法如下。

```
CREATE DATABASE my_db TEMPLATE my_template_db;
```

你可以使用任何一个现存的 database 作为创建新数据库时的模板。此外，你还可以将某个现存的数据库标记为模板数据库，对于这种被标记为模板的数据库，PostgreSQL 会禁止对其进行编辑或者删除。任何一个具备 `CREATEDB` 权限的角色都可以使用这种模板数据库。以超级用户身份运行以下 SQL 可使任何数据库成为模板数据库。

```
UPDATE pg_database SET datistemplate = TRUE WHERE datname = 'mydb';
```

如果你希望修改或者删除被标记为模板的数据库，请先将上述语句中的 `datistemplate` 字段值改为 `FALSE`，这样就可以放开编辑限制。如果你还希望此数据库作为模板的话，修改完后记得将此字段值改回来。

2.4.2 schema的使用

schema 可以对 database 中的对象进行逻辑分组管理。如果你的服务器上有一堆的 database，那么管理起来会很麻烦，可以考虑通过 schema 来对数据进行分类并全部存放到一个 database 中。schema 中的对象名不允许重复，但同一个 database 的不同 schema 中的对象是可以重名的。如果你将数据库中所有表都塞到 `public` schema 中（建数据库时默认创建的 schema），迟早会遇到对象重名的问题。你可以自行决定如何管理和组织 schema。例如：假设要为一家航空公司设计 IT 系统，那么可以将飞机信息表及其日常维护信息表放到一个叫作 plane 的 schema 中，把所有机组人员及其人事信息放到人事 schema 中，再创建一个单独 schema 用于记录乘客相关的信息，这样就把所有信息分门别类隔离开了。

另外一种常见的管理 schema 的方法是基于角色的管理。当系统拥有多个客户端并且每个客户端的数据必须完全隔离时，这种方法特别合适。

假设你的工作是开发一套“宠物狗信息管理系统”并将该在线系统租赁给宠物狗 SPA 店使用。通过广告，现在你有了一些客户，但该系统的数据库中目前仅用了一张 `dogs` 表来存储所有宠物狗的信息。你的系统前期已经满足了政府要求的一堆稀奇古怪的规定，但还有一个要求没达到，那就是客户间数据必须完全隔离，即必须得保证一家 SPA 店看不到另一家 SPA 店的宠物狗信息。为了达到这个要求，你可以为每家客户都建立一个单独的 schema，每个 schema 中建立相同的一张 `dogs` 表。然后就可以把这些宠物狗的数据从单一的 `dogs` 表分散到不同 schema 的 `dogs` 表中。最后为每个 schema 创建一个与之同名的可登录角色，这样就可以实现各自独立管理：`doggy_day_care schema` 归 `doggy_day_care` 角色管理，而 `hot_dogs schema` 归 `hot_dogs` 角色管理，以此类推。所有的宠物狗信息现在完全被分散到它们对应的 SPA 店的 schema 下。当店家的客户端登录到你的数据库并进行数据编辑时，每家店就只能访问他们各自 schema 中的数据。

别急，到这儿还没完，后面还有更妙的用法。我们之前让角色和 schema 同名，这样就可以用上另外一种很有用的技巧，在介绍这个技巧之前需要先介绍一下 search_path 这个系统变量。

我们之前已经说过，schema 中的对象是不允许重名的，但不同 schema 中的对象就可以。例如，在所有 12 个 schema 中都有一张同名的 dogs 表。那么问题来了，当执行类似 SELECT * FROM dogs 这种语句时，PostgreSQL 是怎么知道要查的是哪个 schema 中的表呢？这个问题最简单的解决办法是在表名前加上所属 schema 的名称，比如 SELECT * FROM doggy_day_care.dogs；另一种方法是通过设置 search_path 变量来解决，比如可以设定为：public,doggy_day_care,hot_dogs。当执行查询语句时，规划器会按照从 public schema 到 doggy_day_care schema 再到 hot_dogs schema 的顺序来寻找 dogs 表。

PostgreSQL 有一个少为人知的系统变量叫作 user，它代表了当前登录用户的名称。执行 SELECT user 就能看到其名称。

我们前面将 schema 的名称取得和登录用户名一致，现在可以充分利用这一点了，接下来在 postgresql.conf 中将 search_path 变量设成下面这样。

```
search_path = "$user", public;
```

好了，如果当前登录的角色是 doggy_day_care，那么所有的查询都会优先去 doggy_day_care schema 中寻找目标表，如果找不到才会去 public schema 下找。最重要的一点是，这样我们系统中的 SQL 语句就只需要一种写法，而不用在每个客户的 SQL 中加上对应的 schema 名。这样就算你的客户数增长到几千个甚至是几十万个也没关系，系统中所有的 SQL 语句都不需要修改。为了让整个业务体系进一步简化，你可以建立一个空模板数据库，这样新增客户的时候只需要为此客户执行简短几个步骤即可：创建好 schema、database、角色以及空的业务表就行了。

我们强烈推荐为每一个扩展包创建一个单独 schema 来容纳其对象。安装一个新的扩展包时，会在数据库服务器上创建大量的表、函数、数据类型以及其他对象。默认情况下它们都会被安装到 public schema 中，这样日积月累之后 public schema 里面会被搞得一团糟。例如，完整的 PostGIS 扩展包安装后会创建超过 1000 个函数，如果你此前已经在 public schema 中创建了一些自己的表和函数，可以想象一下，在加进来这上千个表和函数后，要从中找到属于你自己的那些是多么痛苦的一件事情！

在安装扩展包之前，先为其创建一个 schema。

```
CREATE SCHEMA my_extensions;
```

然后把这个新的 schema 加入 search_path：

```
ALTER DATABASE mydb SET search_path='"$user", public, my_extensions';
```

安装扩展包时，记得在 `CREATE EXTENSION` 语句中将你为其创建的新 schema 声明为其归属 schema。



对于现有连接来说，`SET search_path` 命令执行后是不能直接生效的，你需要断开此连接并重连才可以。

2.5 权限管理

PostgreSQL 的权限管理机制非常灵活而自由，因此要想管理得当是很需要一些技巧的。比如，权限控制可精确到数据库对象级别，如有必要甚至可以针对同一张表的不同字段分别单独设定其权限。要想完整地介绍所有关于权限管理的知识可能会需要好几章的篇幅，因此我们在本节中仅介绍能让你达到正常使用程度所必备的知识，同时会指导你避开一些隐蔽的“雷区”，这些“雷”一旦踩到，会导致要么你根本无法访问想要访问的内容，要么服务器上的数据得不到有效防护。

请参考官方手册中“权限管理”章节（http://www.postgresql.org/docs/current/interactive/ddl-priv.html）来了解权限管理体系的概要。

做好 PostgreSQL 的权限管理可不是件很轻松的活。利用 pgAdmin 工具的图形化界面来进行操作会简单一些，或者说至少能让你比较清楚地了解到系统当前权限设置的全貌。通过 pgAdmin 可以完成绝大多数权限管理工作。如果你得负责权限管理工作而你又是个 PostgreSQL 新手，那么建议使用这个工具。如果等不了我们按部就班的慢慢介绍，你也可以直接跳到 4.2.3 节去学习。

2.5.1 权限的类型

PostgreSQL 中支持的对象级权限包括 `SELECT`、`INSERT`、`UPDATE`、`ALTER`、`EXECUTE`、`TRUNCATE` 等以及一个附带的 `WITH GRANT` 修饰符。除了 `GRANT` 外，前几类权限都可顾名思义猜到其含义，`GRANT` 的用法在 2.5.3 节中会专门介绍。请注意，每种权限都有其适用的数据库资产类型，比如对于函数来说 `TRUNCATE` 权限毫无意义，对表来说 `EXECUTE` 权限也无意义。

2.5.2 入门介绍

假设你已安装好 PostgreSQL，建好了一个超级用户角色并设定好了密码。请参照以下步骤来建立其他角色并设定其权限。

(1) PostgreSQL 在安装阶段会默认创建一个超级用户角色以及一个 database，二者的名称都

是 postgres。请以 postgres 身份登录服务器。

(2) 在创建你自己的首个 database 之前，需要先建一个角色作为此 database 的所有者，所有者可以登录该库。语法如下：

```
CREATE ROLE mydb_admin LOGIN PASSWORD 'something';
```

(3) 创建 database 并设定其所有者：

```
CREATE DATABASE mydb WITH owner = mydb_admin;
```

(4) 然后用 mydb_admin 身份登录并创建 schema 和表。

2.5.3 GRANT

GRANT 命令可以将权限授予他人。基本用法如下。

```
GRANT some_privilege TO some_role;
```

请牢记以下几条关于 GRANT 的使用原则。

- 只有权限的拥有者才能将权限授予别人，并且拥有者自身还得有 GRANT 操作的权限。这一点是不言而喻的，因为自己没有的东西当然给不了别人。
- 有些权限只有对象的所有者才能拥有，任何情况下都不能授予别人。这类权限包括 DROP 和 ALTER。
- 对象的所有者天然拥有此对象的所有权限，不需要再次授予。
- 授权时可以加上 WITH GRANT 子句，这意味着被授权者可以将得到的权限再次授予别人。示例如下。

```
GRANT ALL ON ALL TABLES IN SCHEMA public TO mydb_admin WITH GRANT OPTION;
```

- 如果希望一次性将某个对象的所有权限都授予某人，可以使用 ALL 关键字，而不需要一个个权限都写下来[2]。

```
GRANT ALL ON my_schema.my_table TO mydb_admin;
```

- ALL 关键字还可以用于指代某个 database 或者 schema 中的所有对象。

```
GRANT SELECT, UPDATE ON ALL SEQUENCES IN SCHEMA my_schema TO PUBLIC;
```

- 如果希望将权限授予所有人，可以用 PUBLIC 关键字来指代所有角色。

```
GRANT USAGE ON SCHEMA my_schema TO PUBLIC;
```

注 2：原文此处示例不妥，我做了修改。——译者注

官方手册的“GRANT”章节（http://www.postgresql.org/docs/current/interactive/sql-grant.html）中对 GRANT 命令的所有细 节都有极其详尽的说明，我们强烈推荐你先认真阅读一下此章节，以免不小心设错权限导致系统安全隐患。

默认情况下会将某些权限授予 PUBLIC。这些权限包括：CONNECT、CREATE TEMP TABLE（针对数据库）、EXECUTE（针对函数）以及 USAGE（针对语言）。有些情况下出于安全考虑，你可能希望取消一些默认权限，那么可以使用 REVOKE 命令：

```
REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA my_schema FROM PUBLIC;
```

2.5.4 默认权限

从 PostgreSQL 9.0 版开始引入了默认权限，使用默认权限，用户可以一次性针对某个特定 schema 或 database 中的所有数据库资产进行权限设置操作，哪怕这些资产还没创建。如果你的默认权限更新及时，那么这样可以大大简化权限管理工作。

假设我们希望对所有数据库用户都授予某 schema 中所有函数和表的 EXECUTE 和 SELECT 权限，那么我们可以按示例 2-6 这样来定义权限。

示例 2-6：定义 schema 的默认权限

```
GRANT USAGE ON SCHEMA my_schema TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT SELECT, REFERENCES ON TABLES TO PUBLIC;

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT ALL ON TABLES TO mydb_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT SELECT, UPDATE ON SEQUENCES TO public;

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT ALL ON FUNCTIONS TO mydb_admin WITH GRANT OPTION;

ALTER DEFAULT PRIVILEGES IN SCHEMA my_schema
GRANT USAGE ON TYPES TO PUBLIC;
```



新增或者修改默认权限并不会影响已有的权限设置，即只有当某个对象的某项权限未专门设定的情况下，默认权限设定才会生效。

要了解更多关于默认权限的信息，请参考官方手册中“修改默认权限”这一节（http://www.postgresql.org/docs/current/interactive/sql-alterdefaultprivileges.html）的内容。

2.5.5 PostgreSQL权限体系中一些与众不同的特点

最后，在你自己去深入学习了解权限管理体系之前，我们会给你列举一些比较隐蔽的“奇葩”特性。

在一般的数据库中，一个 database 的所有者会对此库中的所有对象都拥有完全的控制权，但 PostgreSQL 不一样，一个 database 的所有者仅对自己在本库中所创建的对象拥有控制权，对其他角色在本库中所创建的对象却没有访问权限。矛盾的是，所有者却又有权能删掉整个库。比如另一个角色在你的库中创建了某个对象，你虽然身为此库的所有者却无权访问这些对象，然后此时你却可以把整个库都删掉。总体来看，这种情况是比较奇怪的。

人们常常会忘记执行 `GRANT USAGE ON SCHEMA` 或者 `GRANT ALL ON SCHEMA` 语句来为 schema 对象进行使用授权。但其实这个步骤是必要的，因为在 PostgreSQL 中，即使已经将 schema 中的表和函数的访问权限授予某个角色，在没有此 schema 的 `USAGE` 权限的情况下，此角色依然不能访问此 schema 中的表或者函数对象。

2.6 扩展包机制

扩展包（extension）是一种用于扩展 PostgreSQL 系统功能的插件机制，该机制的前身被称为“contrib”[3]。该机制很好地体现了开源界的强大优势：人们互相协作、共同开发并自由分享新的功能特性。自从 9.1 版开始引入对 extension 扩展包机制的支持以来，目前它已经发展得非常成熟，这使得为 PostgreSQL 添加功能插件变得非常方便快捷。



对于在 extension 扩展包机制推出之前就已存在的那些历史插件，理论上我们应称之为“contrib”以示差别，但放眼未来，这些老的 contrib 势必都会被改造为用 extension 机制实现。为了描述方便，下文会将二者统称为“扩展包”，但你应该清楚二者的区别。

对于一台 PostgreSQL 服务器来说，并不是其中每个 database 都要安装全部的扩展包，只有当某个 database 的确需要此扩展包提供的功能时才应安装。如果你的 PostgreSQL 服务器上的所有 database 都需要某些扩展包的功能，那么可以新建一个模板数据库（有关模板数据库的介绍，请参见 2.4.1 节），然后在此模板数据库中预先安装好这些扩展包，那么后续的 database 就能以此模板数据库为基础来创建，这样就避免了每新建一个 database 就需要再安装一遍扩展包的麻烦。

注 3：extension 与 contrib 本质上都是 PostgreSQL 的系统功能插件，二者功能类似但实现机制有很大差异，其版本分界点为 9.1 版，之前的插件机制被称为 contrib，9.1 版及之后的插件机制被称为 extension。——译者注

建议定期检查并卸载掉已经不再需要的扩展包以避免系统过于臃肿，因为有的扩展包需要占用相当大的空间。

可以通过示例 2-7 中的语句来查看系统中已经安装了哪些扩展包。你的查询结果可能和我们在下面列出的很不一样，这是正常的，因为每台数据库服务器的具体情况可能都不一样。

示例 2-7：服务器上已安装的扩展

```
SELECT name, default_version, installed_version, left(comment,30) As comment
FROM pg_available_extensions
WHERE installed_version IS NOT NULL
ORDER BY name;

     name      |  def  | installed |                    com
---------------+-------+-----------+-------------------------------------------
 btree_gist    | 1.0   | 1.0       | support for indexing common datatypes in..
 fuzzystrmatch | 1.0   | 1.0       | determine similarities and distance betw..
 hstore        | 1.2   | 1.2       | data type for storing sets of (key, valu..
 plpgsql       | 1.0   | 1.0       | PL/pgSQL procedural language..
 plv8          | 1.3.0 | 1.3.0     | PL/JavaScript (v8) trusted procedural la..
 postgis       | 2.1.3 | 2.1.3     | PostGIS geometry, geography, and raster ..
 www_fdw       | 0.1.8 | 0.1.8     | WWW FDW - extension for handling differe..
```

如果想要了解系统中某个已安装的扩展包的更多详细内容，请在 psql 中执行类似以下的命令：

```
\dx+ fuzzystrmatch
```

或者执行以下查询也可以：

```
SELECT pg_catalog.pg_describe_object(d.classid, d.objid, 0) AS description
FROM pg_catalog.pg_depend AS D INNER JOIN pg_catalog.pg_extension AS E
ON D.refobjid = E.oid
WHERE D.refclassid = 'pg_catalog.pg_extension'::pg_catalog.regclass AND deptype
= 'e' AND E.extname = 'fuzzystrmatch';
```

查询结果显示了该扩展包中包含了哪些内容：

```
description
----------------------------------------------------------------------------
function dmetaphone_alt(text)
function dmetaphone(text)
function difference(text,text)
function text_soundex(text)
function soundex(text)
function metaphone(text,integer)
function levenshtein_less_equal(text,text,integer,integer,integer,integer)
function levenshtein_less_equal(text,text,integer)
function levenshtein(text,text,integer,integer,integer)
function levenshtein(text,text)
```

扩展包中可以包含各类数据库资产，包括：函数、表、数据类型、数据类型转换器、编程语言、运算符类，等等。但函数通常被认为是有效负载的大部分。

2.6.1 扩展包的安装

将扩展包安装到系统中需要两个步骤：首先，下载安装包并安装到数据库服务器上；其次，将此扩展包安装到目标 database 中。



我们在前述两个步骤中都使用了“安装”这个词，但其指代的具体动作是不一样的，当上下文环境不清楚时，我们会加以描述区分。

以下我们将介绍扩展包的安装方法，同时也将介绍在不支持 extension 扩展包机制的老版本 PostgreSQL 上安装 contrib 扩展包的方法。

1. 步骤一：将扩展包安装到数据库服务器

这一步的具体做法会根据操作系统的不同而有所不同。总的来说就是先下载该扩展包的安装文件以及该扩展包所依赖的库文件，然后将它们分别复制到操作系统的 bin 和 lib 文件夹，同时把 SQL 脚本文件复制到 share/extension 文件夹（9.1 版及之后版本）或者 share/contrib 文件夹（9.1 版之前的版本）。这样就为接下来执行第二步做好了准备。

对于较小的扩展包来说，其所需的很多库文件在 PostgreSQL 安装好以后就有了，或者没有的话也可以通过 `yum` 或 `apt get postgresql-contrib` 命令较容易地获取到。对于通过以上方式获取不到的库文件，你要么自行编译，要么找一下别人已经编译好的安装包，要么从另一台环境完全相同的服务器上把库文件复制过来。对于 PostGIS 这类较大的扩展包，通常可以从你下载 PostgreSQL 的站点下载到完整安装包。如果想了解当前服务器上有哪些扩展包可用，请执行以下命令：

```
SELECT * FROM pg_available_extensions;
```

2. 步骤二：将扩展包安装到数据库中（9.1版之前的做法）

在 9.1 版之前，安装 contrib 扩展包时需要在数据库上手工执行一些 SQL 脚本。一般来说，如果你下载的扩展包是一个可安装的程序，那么执行此程序后会自动将附加的脚本转储到 PostgreSQL 安装路径下的 contrib 文件夹中。该路径的具体位置可能会随着操作系统和 PostgreSQL 版本的不同而有所不同。

例如，对于 CentOS 上的 PostgreSQL 9.0 版来说，执行 pgAdmin 扩展包安装脚本的命令行如下：

```
psql -p 5432 -d postgres -f /usr/pgsql-9.0/share/contrib/adminpack.sql
```

该命令会以非交互方式调用 psql 来执行一个 SQL 脚本。

contrib 与 extension 还有一个重要差别，那就是 contrib 机制不支持对扩展包信息进行查询，比如查询已安装哪些扩展包以及有哪些安装包可安装等。

3. 步骤二：将扩展包安装到数据库中（9.1版及之后版本的做法）

9.1 版引入的 extension 扩展包机制使得安装过程更加简单和连贯。使用 `CREATE EXTENSION` 命令即可将扩展包安装到指定的 database 中。相比原来的安装方法，该新机制有三大主要优点：首先，用户不需要弄清楚扩展包文件存放的具体路径（share/extension）；其次，可以通过 `DROP EXTENSION` 命令方便地卸载扩展包；最后，支持查看当前已安装和可安装的扩展包列表。PostgreSQL 安装包中已经附带了最常用的若干扩展包，因此安装时你仅需执行 `CREATE EXTENSION` 命令来安装即可。通过 PostgreSQL Extension Network 站点（http://pgxn.org/）可以下载到 PostgreSQL 安装包中默认未附带的扩展包。

以下是安装 fuzzystrmatch 扩展包的命令：

```
CREATE EXTENSION fuzzystrmatch;
```

你仍可以使用 psql 以非交互方式安装扩展包。先连接到需要安装此扩展包的 database，然后执行类似以下命令行：

```
psql -p 5432 -d mydb -c "CREATE EXTENSION fuzzystrmatch;"
```



基于 C 语言的扩展包必须由具备超级用户权限的角色来安装。大多数扩展包都是基于 C 语言的。

我们建议你创建专门的 schema 来安装扩展包，以确保扩展包数据与业务数据隔离。建好 schema 后，执行以下命令来将其指定给待安装的扩展包：

```
CREATE EXTENSION fuzzystrmatch SCHEMA my_extensions;
```

4. 升级PostgreSQL版本以支持新的extension扩展包机制

如果你从 PostgreSQL 9.1 版或者更早的版本升级到了 9.1 版或者之后的版本，并且也正确地将数据从老版本导入到了新版本之中，那么之前安装的那些 contrib 扩展包依然是可以正常工作的。但为了简化管理并提升可维护性，你应该在 contrib 文件夹中升级你的旧扩展包以对扩展包使用新的方法。这种格式升级是完全可以实现的，尤其是对那些随 PostgreSQL 版本附带的扩展包来说更加没问题。不过请注意，这里说的“升级”是指从 contrib 格式到 extension 格式的“格式升级”，而不是指扩展包本身的“功能升级”。

例如，如果你之前在一套 PostgreSQL 9.0 版的 `contrib` schema 中安装了 tablefunc 扩展包（这是一个用于实现跨表查询的功能插件），然后你将该系统升级到了 9.1 版。那么可以执行以下命令来升级此插件：

```
CREATE EXTENSION tablefunc SCHEMA contrib FROM unpackaged;
```

该命令会搜索 `contrib` schema，找到所有属于 tablefunc 插件的成员对象，然后把它们按照新的 extension 扩展包模型格式进行封装，这样该扩展包就会出现在 `pg_available_extensions` 视图信息中，就好像是全新安装的一个 extension 扩展包一样。这样就实现了从 contrib 到 extension 的扩展包格式升级。

该命令会将 `contrib` schema 中的插件函数按 extension 模式进行封装格式升级，函数本身不会有任何改动，但此后该 database 进行备份时不会包含这些函数，因为它们的身份已经发生了变化，从与别的函数身份无法区分的普通函数变为 `extension` 扩展包中的函数。

你仍可以使用 psql 在某个数据库中安装扩展包，而不必首先连接到该数据库。

```
psql -p 5432 -d mydb -c "CREATE EXTENSION fuzzystrmatch;"
```

2.6.2 通用扩展包

通用扩展包是指那些因功能比较基本和通用而在 PostgreSQL 安装包中默认附带的一些扩展包，但它们不一定会被默认安装，具体视其功能而定。有些早期的通用扩展包已被 PostgreSQL 内核接纳从而“登堂入室”成为系统基础功能，因此如果你是从较老版本的 PostgreSQL 升级上来的话，可能会发现原先要通过安装扩展包才能实现的功能现在已成了系统默认提供的功能。

1. 比较常用的扩展包介绍

从 9.1 版开始，PostgreSQL 官方推荐开发人员使用 extension 扩展包模式来为系统制作功能插件。从仅包含函数和数据类型的基本插件到包含了存储过程语言支持（PL）、索引类型以及外部数据封装器的高级插件，都可以用 extension 扩展包机制来实现。本节我们列举了最常用（也有些人称之为“必备”）但默认情况下 PostgreSQL 并未安装的一些扩展包。你会发现下面列出的很多扩展包在你的 PostgreSQL 系统中已经有了，具体哪些会有哪些没有取决于你使用的是哪个 PostgreSQL 发行版，不同发行版之间可能会略有差异。

- btree_gist（http://www.postgresql.org/docs/current/interactive/btree-gist.html）
  该扩展包实现了基于 B- 树索引算法的 GiST 索引运算符类，适用于 B- 树索引支持的所有数据类型，其具体效果与标准的 B- 树索引类似。更多细节请参见 6.3.1 节的内容。

- btree_gin（http://www.postgresql.org/docs/current/interactive/btree-gin.html）
  该扩展包实现了基于 B- 树索引算法的 GIN 索引运算符类，适用于 B- 树索引支持的所

有数据类型，其具体效果与标准的 B- 树索引类似。更多细节请参见 6.3.1 节的内容。

- postgis（http://postgis.net/）
  该扩展包将 PostgreSQL 提升为一个业界最先进的空间数据库，胜过了所有类似的商业化产品。如果你需要处理标准的 OGC GIS 数据，或者人口统计学数据，又或者地理编码数据，那么 postgis 会是你的必备之选。我们编写的图书 *PostGIS in Action*（http://www.postgis.us/）中对 PostGIS 做了更加详尽的介绍。PostGIS 是扩展包界的“巨无霸”，它包含 800 多个函数、自定义数据类型以及空间索引等对象。

- fuzzystrmatch（http://www.postgresql.org/docs/current/interactive/fuzzystrmatch.html）
  这是一个用于字符串模糊匹配的轻量级扩展包，包含了诸如 `soundex`、`levenshtein` 和 `metaphone` 等函数。我们在“soundex 等模糊匹配函数在 PostgreSQL 中如何实现”这篇文章（http://www.postgresonline.com/journal/archives/158-Where-is-soundex-and-other-warm-and-fuzzy-string-things.html）中介绍了该扩展包的用法。

- hstore（http://www.postgresql.org/docs/current/interactive/hstore.html）
  该扩展包为 PostgreSQL 添加了对键值数据库的支持，同时也支持索引，非常适用于存储非结构化数据。如果你正在寻找一种介于关系型数据库和 NOSQL 数据库之间的产品，可以尝试一下 hstore。

- pg_trgm（trigram，http://www.postgresql.org/docs/current/interactive/pgtrgm.html）
  该扩展包提供了另外一种字符串模糊搜索算法库，可与 fuzzystrmatch 配合使用。在 9.1 版中，该扩展包新增了一种运算符类，使得基于 `ILIKE` 的搜索操作能用上索引。该扩展还能够让形如 `LIKE '%something%'` 的通配符查询能用上索引。关于这部分内容，在“ILIKE 和 LIKE 操作的新技巧介绍”这篇博文（http://www.postgresonline.com/journal/archives/212-PostgreSQL-9.1-Trigrams-teaching-LIKE-and-ILIKE-new-tricks.html）中有更深入的探讨。

- dblink（http://www.postgresql.org/docs/current/interactive/dblink.html）
  该扩展包支持从一台 PostgreSQL 服务器远程访问另一台 PostgreSQL 服务器上的数据。在 9.3 版中引入对外部数据源的支持之前，dblink 是唯一能够实现跨数据库交互的机制。目前该机制一般用于需要临时性连接到外部数据源或者临时的即席查询场景。例如，如果我们需要从备份数据中找回一些被误删的数据，那么我们一般会用 dblink 来连接到依据备份的数据恢复出来的老版本数据库。

- pgcrypto（http://www.postgresql.org/docs/current/interactive/pgcrypto.html）
  该扩展包提供了一系列的加密工具，包括使用广泛的 PGP 算法。使用该扩展包提供的功能来对数据库中存储的信用卡号码或其他顶级机密信息进行加密是非常方便的。在“使用 pgcrypto 扩展包来实现数据加密”这篇博文（http://www.postgresonline.com/

journal/archives/165-Encrypting-data-with-pgcrypto.html）中我们对其用法进行了快速的入门介绍。

2. 经典扩展包介绍

此处我们介绍两个经典扩展包，之所以称其为“经典”，是因为它们使用非常广泛并因此被 PostgreSQL 官方接纳而成为了系统内核功能的一部分，但在老版本的 PostgreSQL 上它们还是以扩展包的形式存在的，你在实际工作中有可能遇到这些老版本，所以我们在此还是有必要介绍一下这些曾经的扩展包。

- tsearch（http://www.postgresql.org/docs/current/interactive/textsearch-intro.html）
  该扩展包封装了一系列用于强化全文搜索功能的索引、运算符、自定义词典和函数。目前该扩展包的功能已被系统内核接纳并成为 PostgreSQL 基础功能的一部分。如果你使用的 PostgreSQL 版本较老，在其上 tsearch 还是以扩展包的形式存在，那么我们建议你可以升级到 tsearch2 这个新版本。当然，最好的做法其实是将 PostgreSQL 服务器软件升级到新版本，因为老版本上的 tsearch 插件的兼容性支持随时可能终止，这会导致你在老版本的 PostgreSQL 上再也无法享受到相关的功能更新和 BUG 修复。

- xml（http://www.postgresql.org/docs/current/interactive/functions-xml.html）
  该扩展包提供了对 XML 数据类型的支持以及相关的函数和运算符。为了达到 ANSI SQL XML 标准的要求，PostgreSQL 内核接纳了该插件包的部分功能。其余未被纳入内核的那部分功能仍以扩展包的形式存在，不过已改名为 xml2。具体来说，如果你需要使用 `xlst_process` 函数来处理 XSL 模板数据，那么就需要安装 xml2 扩展包。另外 xml2 扩展包中还含有一些 XPath 函数。

2.7 备份与恢复

PostgreSQL 自身附带了两个备份工具：`pg_dump` 和 `pg_dumpall`，二者均位于 bin 文件夹下。`pg_dump` 可备份一个指定的 database，而 `pg_dumpall` 可一次性备份所有 database 的数据以及系统全局数据。由于 `pg_dumpall` 需要能够访问系统中的所有 database，因此必须由具备 superuser 权限的角色来执行。这两个工具的大多数命令行选项都可以有两种写法：一种是 GNU 风格（两个横杠连字符后跟一个单词）；另一种是传统的单字母风格（一个横杠连字符后跟一个字母）。这两种写法是完全等价的，甚至在同一个命令行中也可以混用。本节中我们仅讨论一些基本的用法，如果要了解更多深入的内容，请参考 PostgreSQL 官方手册中的“备份与恢复”（http://www.postgresql.org/docs/current/interactive/backup.html）。

在学习本节内容时，你会发现我们一般会在示例命令行中指明目标数据库所在的主机地址和监听端口，这是因为我们一般是在另外一台机器上通过执行定时任务（通过 `pg_agent` 实现）的方式来执行备份，这种情况下必须在命令行写明目标数据库的地址和侦听端口；或者另外一种情况是在同一台机器上运行多个 PostgreSQL 服务实例，实例间的侦听端口互不

相同，所以命令行中必须明确指定 IP 和端口。有一个情况请注意：如果你的服务被设置为仅监听 `local`，那么在备份命令行中加上 `-h`（或者是 `--host`）选项并带上主机 IP 反而会引发问题，因为默认情况下只有来自 localhost 的连接才会被允许接入，指定 IP 反而会导致连接失败。如果备份任务是直接在数据库服务器所在的机器上执行的，那么命令行中可以根本不用指定主机地址，这样可以避免出问题。

`pg_dump` 和 `pg_dumpall` 工具不支持在命令行选项中设定登录密码，因此为了便于执行自动任务，你需要在 postgres 操作系统账号的 home 文件夹下创建一个密码文件 .pgpass 来存储密码；或者也可以用 `PGPASSWORD` 环境变量来设定密码。

2.7.1　使用pg_dump进行有选择性的备份

如果你希望每天都进行备份，那么使用 `pg_dump` 比 `pg_dumpall` 更合适，因为前者支持精确指定要备份的表、schema 和 database，而后者不支持。`pg_dump` 可以将数据备份为 SQL 文本文件格式，也支持备份为用户自定义压缩格式或者是 TAR 包格式。在数据恢复时，对压缩格式和 TAR 包格式的备份文件可以实现并行恢复，该特性是从 8.4 版开始支持的。我们认为 `pg_dump` 是你进行日常备份时不可或缺的工具，因此我们在本书附录 B 的 B.1 节中提供了一张完整的 `pg_dump` 帮助信息清单，这样你就可以对其数量众多的选项开关用法一目了然。

下面的例子展示了一些常见的备份场景以及相应的 `pg_dump` 选项。这些例子对于任何版本的 PostgreSQL 应该都是适用的。

备份某个 database，备份结果以自定义压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -f mydb.backup mydb
```

备份某个 database，备份结果以 SQL 文本方式输出，输出结果中需包括 `CREATE DATABASE` 语句：

```
pg_dump -h localhost -p 5432 -U someuser -C -F p -b -v -f mydb.backup mydb
```

备份某个 database 中所有名称以“pay”开头的表，备份结果以自定义压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -t *.pay* -f pay.backup mydb
```

备份某个 database 中 `hr` 和 `payroll` 这两个 schema 中的所有数据，备份结果以自定义压缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -n hr -n payroll -f hr.back-
up mydb
```

备份某个 database 中除了 `public` schema 中的数据以外的所有数据，备份结果以自定义压

缩格式输出：

```
pg_dump -h localhost -p 5432 -U someuser -F c -b -v -N public -f all_sch_ex
cept_pub.backup mydb
```

将数据备份为 SQL 文本文件，且生成的 INSERT 语句是带有字段名列表的标准格式，该文件可用于将数据导入到低于当前版本的 PostgreSQL 或者其他支持 SQL 的非 PostgreSQL 数据库中（之所以能够实现这种数据移植过程，是因为标准的 SQL 文本可在任何支持 SQL 标准的数据库中执行）：

```
pg_dump -h localhost -p 5432 -U someuser -F p --column-inserts -f se
lect_tables.backup mydb
```



如果输出文件路径中含空格或者其他可能影响命令行正常处理的字符，请在路径两侧加上双引号，比如："/path with spaces/mydb.backup"。请注意这在 PostgreSQL 中是一个通用的原则，即当你不确定某段文本是否能正常处理时，都可以加双引号。

从 9.1 版开始支持目录格式选项。该选项会将每个表备份为某个文件夹下的一个单独的文件，这样就解决了以其他备份格式备份时可能存在的单个文件大小超出操作系统限制的问题。该选项是生成多个文件的唯一 pg_dump 备份格式选项，具体语法参见示例 2-8。备份时会先创建一个新目录，然后逐个表将一个 gzip 格式的压缩文件和一个列出所有包含结构的文件填充到该目录中。如果备份开始时发现指定的目录已存在，那么该命令会报错并退出。

示例 2-8：目录格式备份

```
pg_dump -h localhost -p 5432 -U someuser -F d -f /somepath/a_directory mydb
```

从 9.3 版开始支持并行备份选项 --jobs（-j）。如果将其设定为 --jobs=3，则后台会有三个线程并行执行当前备份任务。此选项只有在按目录格式进行备份时才会生效，每个写线程只负责写一个单独的文件，因此一定是输出结果为多个独立的文件时才可以并行。示例 2-9 演示了其用法。

示例 2-9：目录格式并行备份

```
pg_dump -h localhost -p 5432 -U someuser -j 3 -Fd -f /somepath/a_directory mydb
```

2.7.2 使用pg_dumpall进行全库备份

pg_dumpall 工具可以将当前 PostgreSQL 服务实例中所有 database 的数据都导出为 SQL 文本（请注意：pg_dumpall 不支持导出 SQL 文本以外的其他格式），也可以同时导出表空间定义和角色等全局对象。要了解该命令支持的所有选项，请参考本书附录 B 的 B.2 节的内容。

我们建议你每天都对角色和表空间定义等全局对象进行备份，但不建议每天都使用 pg_dumpall 来备份全库数据，因为 pg_dumpall 仅支持导出为 SQL 文本格式，而使用这种庞大的 SQL 文本备份来进行全库级别的数据恢复是极其耗时的，所以一般只建议用 pg_dumpall 来备份全局对象而非全库数据，如果你一定要用 pg_dumpall 来备份全库数据的话，一般一个月执行一次就够了。

以下命令可实现仅备份角色和表空间定义：

```
pg_dumpall -h localhost -U postgres --port=5432 -f myglobals.sql --globals-only
```

如果仅需备份角色定义而无需备份表空间，那么可以加上 --roles-only 选项：

```
pg_dumpall -h localhost -U postgres --port=5432 -f myroles.sql --roles-only
```

2.7.3 数据恢复

PostgreSQL 支持以下两种数据恢复方法：

- 使用 psql 来恢复 pg_dump 或者 pg_dumpall 工具生成的 SQL 文本格式的数据备份；
- 使用 pg_restore 工具来恢复由 pg_dump 工具生成的自定义压缩格式、TAR 包格式或者目录格式备份。

1. 使用psql恢复SQL文本格式的数据备份

所谓的 SQL 文本格式的数据备份其实就是一个包含 SQL 脚本的文本文件。这种备份格式使用起来最不方便，但却是最通用的一种格式。恢复时需要将此 SQL 脚本全部执行一遍。你无法选择性地仅恢复部分数据，除非你手工编辑此文件。以下示例都是在操作系统命令行界面执行，过程中可能需要在 psql 界面上输入密码。

恢复一个 SQL 备份文件并忽略过程中可能发生的所有错误：

```
psql -U postgres -f myglobals.sql
```

恢复一个 SQL 备份文件，如遇任何错误则立即停止恢复：

```
psql -U postgres --set ON_ERROR_STOP=on -f myglobals.sql
```

将 SQL 文本中的数据恢复到某个指定的 database：

```
psql -U postgres -d mydb -f select_objects.sql
```

2. 使用pg_restore进行恢复

如果你是使用 pg_dump 进行的备份，并且选择的输出格式为 tar（TAR 包格式）或者 custom（自定义压缩格式）或者 directory（目录格式，为每张表生成一个文件，输出到指定路径中），那么你可以使用功能强大的 pg_restore 工具来进行恢复。pg_restore 支持的

选项之多令人眼花缭乱，远远超过我们所使用过的其他任何数据库中提供的恢复工具。以下介绍一些该工具的特别功能。

- 支持并行恢复，使用 -j 选项可以控制并行恢复的线程数。多个恢复线程可以并行处理，每个线程处理一张表。该模式可以显著提高恢复速度。
- 你可以使用 pg_restore 扫描备份文件来生成一张备份内容列表，通过该列表可以确认备份中包含了哪些内容。你还可以通过编辑该内容列表来控制恢复哪些内容。
- pg_dump 支持选择性地仅备份部分对象以节省备份时间，类似地，pg_restore 也支持选择性地仅恢复部分对象，不管备份文件本身是全库备份还是部分对象的备份都没有问题。
- pg_restore 的大部分功能是后向兼容的，即支持将老版本 PostgreSQL 生成的备份数据恢复到新版本的 PostgreSQL 中。

如想了解 pg_restore 命令所支持的全部选项，请参考附录 B 中 B.3 节的内容。

在使用 pg_restore 执行恢复动作之前，请先创建目标数据库：

```
CREATE DATABASE mydb;
```

然后执行恢复：

```
pg_restore --dbname=mydb --jobs=4 --verbose mydb.backup
```

如果备份和恢复时使用的 database 同名，则可以通过加 --create 选项省去单独建库的过程，命令如下：

```
pg_restore --dbname=postgres --create --jobs=4 --verbose mydb.backup
```



如果指定了 --create 选项，那么恢复出来的数据库名就会默认采用备份时的数据库名，不允许改名。如果还同时指定了 --dbname 选项，那么此时连接的数据库名一定不能是待恢复的数据库名，因为要恢复数据库之前必然要建数据库，而要建数据库之前必然要先连到某个已存在的数据库，--dbname 选项指定的就是建立被恢复的数据库之前先连到哪个数据库，所以必然不能与待恢复的数据库重名，我们一般指定为先连到 postgres 数据库。

9.2 版或更新版本的 pg_restore 支持 --section 选项，加上该选项后可以实现仅恢复表结构而不恢复表数据。当我们希望创建模板数据库时可以用这个方法，因为模板数据库一般不需要带数据。具体做法是先创建恢复目标数据库：

```
CREATE DATABASE mydb2;
```

然后使用 pg_restore：

```
pg_restore --dbname=mydb2 --section=pre-data --jobs=4 mydb.backup
```

2.8 基于表空间机制进行存储管理

PostgreSQL 使用“表空间”这一概念来将逻辑存储空间映射到磁盘上的物理存储空间。PostgreSQL 在安装阶段会自动生成两个表空间：一个是 `pg_default`，用于存储所有的用户级数据；另一个是 `pg_global`，用于存储所有的系统级数据。这两个表空间就位于默认的数据文件夹下。你可以不受限地创建表空间并将其物理存储位置设定到任何一块物理磁盘上。你也可以为 database 设定默认表空间，这样该 database 中创建的任何新对象都会存储到此表空间上。你也可以将现存的数据库对象迁移到新的表空间中。

2.8.1 表空间的创建

创建表空间需要先为其取一个逻辑名称并指定某个物理文件夹作为其存储位置，注意要确保 `postgres` 操作系统账户对此文件夹有完全的访问权限。如果你目前使用的是 Windows 服务器，请使用如下命令行（注意使用 Unix 风格的斜杠作为路径分隔符）：

```
CREATE TABLESPACE secondary LOCATION 'C:/pgdata94_secondary';
```

对于基于 Unix 的系统来说，你必须先创建文件夹或者定义一个 fstab 位置，然后执行以下命令：

```
CREATE TABLESPACE secondary LOCATION '/usr/data/pgdata94_secondary';
```

2.8.2 在表空间之间迁移对象

你可以将数据库中的对象在表空间之间随意迁移。如果希望将一个 database 的所有对象都移动到另一个表空间中，可以执行以下命令：

```
ALTER DATABASE mydb SET TABLESPACE secondary;
```

如果只希望移动一张表，命令如下：

```
ALTER TABLE mytable SET TABLESPACE secondary;
```

PostgreSQL 9.4 版中引入了一个新功能：一次性把一个表空间的多个对象迁移到另一个表空间。如果命令执行者是超级用户，那么源表空间所有的对象都会被迁移过去；否则仅会迁移执行者所拥有的对象。

将 `pg_default` 默认表空间中的所有对象迁移到 secondary 表空间，所需的命令行如下：

```
ALTER TABLESPACE pg_default MOVE ALL TO secondary;
```

在迁移过程中所涉及的 database 和表会被锁定。

2.9 禁止的行为

我们发现人们总是会犯各种各样、花式翻新的错误来将 PostgreSQL 系统搞崩溃，因此在本章的最后一节中，我们认为有必要逐条列出那些最常见的错误。对于初学者来说，如果你搞不清到底哪里出了问题，那么请首先查看系统日志，从中可以找到解决问题的线索。日志文件位于数据文件夹的根目录或者其中的 pg_log 子文件夹下。也有可能系统在记下日志之前即已崩溃，那么显然这种情况下查日志是没用的。如果你的 PostgreSQL 服务启动失败，请尝试执行以下操作系统命令。

```
path/to/your/bin/pg_ctl -D your_postgresql_data_folder
```

2.9.1 切记不要删除PostgreSQL系统文件

你可能觉得这是废话，但当磁盘空间不够的时候有些人就会慌手慌脚地从 PostgreSQL 的数据文件夹下删除文件，因为它占用的空间实在是太大了。出现这种问题的部分原因是有些文件夹的名称的确很容易令人误解，比如：pg_log、pg_xlog 和 pg_clog。这些名称看起来就像是用于存放日志的文件夹，所以可以“安心地”删除。这种做法是危险的，因为有些文件删了没事，有些删了就会导致数据库被破坏。

pg_log 文件夹一般会在 data 文件夹下，其体积可能增长得很快，尤其是当打开了日志开关的时候。这个文件夹下的文件任何时候都可以安全删除，事实上很多人会设置一个定时任务来定期清空这些日志文件。

除了 pg_xlog 文件夹下的文件可以有条件地删除外，其他 PostgreSQL 系统文件夹中的文件都不能删，即使有的文件夹名称中带有 log 字样因而看起来像是某种日志，也绝对不能删，比如 pg_clog 文件夹中存储的是活跃事务提交日志，千万不要碰。

pg_xlog 文件夹用于存储事务日志。我们见过有的系统中会在 pg_xlog 文件夹下建一个子文件夹 archive，专门用于存放归档的事务日志。一般来说你的系统总会需要建一个专门的文件夹（该文件夹不一定要放在 pg_xlog 下）用于日志归档，因为如果这是一个同步复制环境，那么需要持续地进行事务日志归档；需要归档文件夹的另一个理由是得把这些日志存下来以备不时之需，因为我们有可能需要将系统数据恢复到过去的某个时间点。将 pg_xlog 文件夹下的所有文件都删除会导致系统数据被严重破坏，比如引发数据不一致或丢失等。但仅删除归档文件夹下的日志却没这么严重，最多会导致无法恢复到过去的某个时点，或者是在同步环境下可能导致从属服务器无法进行数据同步，因为还未来得及同步的一些日志文件可能已被删掉了。如果以上场景在你的系统中都不涉及，那么可以放心地删除归档文件夹下的日志文件。

要当心某些过于“尽责”的杀毒软件，特别是在 Windows 上。我们曾经遇到过杀毒软件把很重要的 PostgreSQL 可执行文件给删掉的案例。如果在 Windows 环境下发现

PostgreSQL 无法启动，记得首先查看一下事件查看器（event viewer）的记录，其中可能存在有用的线索。

2.9.2 不要把操作系统管理员权限授予PostgreSQL的系统账号（postgres）

很多人可能会误认为 postgres 这个操作系统账号必须拥有操作系统的管理员权限。事实上，在有的 PostgreSQL 版本上，如果给予了 postgres 账号管理员权限，很有可能导致该机器操作系统启动失败。postgres 账号的身份应该就是一个普通用户账号，其权限只要能够访问 data 文件夹以及其他表空间文件夹即可。大多数 PostgreSQL 安装包会自动为 postgres 账号设定够用的权限，请不要再画蛇添足。在 SQL 注入攻击中，那些不必要的权限会为攻击者们提供可乘之机。

有些情况下，的确是有必要把 data 文件夹以外的某些文件夹或者可执行程序的“改写 / 删除 / 读取”权限授予 postgres 账号。比如，当需要设置一个定时任务来执行批处理任务时就需要这么干。针对这种情况，我们也建议你仅仅授予 postgres 账号最小的必要权限，而不应该为了图省事而直接授予其最高权限。

2.9.3 不要把shared_buffers缓存区设置得过大

禁止将 shared_buffers 设置得和系统当前内存总容量一样大，这么做很可能会导致操作系统崩溃或者是启动失败。在 32 位 Windows 系统上，该设置如果超过 512 MB 就可能导致系统出问题。在 64 位 Windows 上，该限制可以放宽到 1 GB 或者更多一些也没问题。在有些 Linux 系统上，不能将 shared_buffers 设置得大于编译过的 SHMMAX 变量，而一般来说 SHMMAX 的值是比较小的，所以这就给 shared_buffers 的设置带来了一些限制。PostgreSQL 9.3 版修改了内核对于内存的使用方法，因此之前版本中那些因内核限制而导致的问题从该版本开始不复存在。官方手册中“内核资源管理”（http://www.postgresql.org/docs/current/interactive/kernel-resources.html）这一节介绍了更多关于这方面的细节。

2.9.4 不要将PostgreSQL服务器的侦听端口设为一个已被其他程序占用的端口

如果启动 PostgreSQL 时系统发现侦听端口已被占用，那么会在 pg_log 中记录类似这样的错误日志：make sure PostgreSQL is not already running。以下是可能导致该问题的常见原因。

- postgres 服务实例已经启动好了，而你正试图再启动一遍。
- PostgreSQL 服务侦听端口已被其他程序占用。

- `postgres` 服务之前发生过异常关闭或者崩溃，在 data 文件夹下遗留了一个 postgresql.pid 文件。请直接删除该文件并再次尝试启动。
- 有一个孤立的 PostgreSQL 进程还在运行。如果前述方法都试过了但问题仍未解决，请终止所有还在运行的 PostgreSQL 进程然后再次尝试启动。

第 3 章

psql工具

psql 是 PostgreSQL 自带的一个不可或缺的命令行执行工具，其用途多种多样，除了执行 SQL 这个基本功能外，还可用于自动化执行脚本、导入导出数据、恢复表数据以及执行数据库管理任务，它甚至还可以作为一个简单的报表生成器来使用。如同其他命令行工具一样，psql 使用时也需要了解各种各样的选项。如果你无法以图形界面工具访问 PostgreSQL，那么 psql 可能是你执行 SQL 语句以及进行数据库管理任务的唯一选择。在学习本章时，我们建议你将本书附录 B 中 B.4 节的 psql 转储的帮助信息打印出来，放在手边以备查阅。

3.1 环境变量

在设置了 `PGHOST`、`PGPORT` 和 `PGUSER` 等环境变量后，在调用 psql 命令行时就可以不用显式地指定主机、端口和用户，系统会自动使用环境变量设定的值，这一点跟 PostgreSQL 自带的其他命令行工具是一样的。关于环境变量的使用细节，请参考官方手册中“环境变量”这一节的介绍（http://www.postgresql.org/docs/current/interactive/libpq-envars.html）。你也可以用 `PGPASSWORD` 这个环境变量来设置登录密码，或者是通过密码文件来设置也可以。有关密码文件的详细信息，请参见官方手册中“密码文件”这一节的介绍（http://www.postgresql.org/docs/current/interactive/libpq-pgpass.html）。从 PostgreSQL 9.2 版开始，psql 开始支持以下两个新的环境变量。

- `PSQL_HISTORY`

  该变量用于设置 psql 历史日志文件名，该日志中记录了近期通过 psql 执行过的所有命令行，其默认值为 ~/.psql_history。

- `PSQLRC`

  该变量用于设置配置文件的路径和文件名。

如果你既未设定相应的环境变量也未在命令行中指定相关形参，那么 psql 会使用系统默认值。在本章示例中，我们假定你使用默认值或已设置这些变量。如果你使用 pgAdmin 工具，那么也可以通过其工具栏上的“插件”菜单项来直接打开 psql（详见 4.2.1 节的内容），点击后会打开一个已经连接到相应 database 的 psql 窗口。

3.2 psql的两种操作模式：交互模式与非交互模式

直接在操作系统的命令行界面上键入 `psql` 并回车，从操作系统提示符切换到 psql 提示符后就表示已经进入了 psql 的交互模式界面。你现在就可以执行命令了，记得要输入分号作为命令结束标记，要是不输就直接回车的话，psql 会认为命令还未输入完成，会在换行后等待继续输入。

在 psql 界面上键入 `\?` 会列出交互模式下支持的所有命令。为了方便读者，我们将这些命令列表放在本书附录 B 中，并对一些最新版本中添加的条目以高亮标记，具体请参见本书附录 B 中的 B.4 节。如果在 psql 界面上键入 `\h` 后跟命令关键字，则会打印出该命令在 PostgreSQL 官方手册中相应的语法帮助信息。

要在非交互模式下使用 psql，请从 OS 命令提示符执行 `psql`，并给其传送一个脚本文件。psql 的“非交互模式”是指在调用 psql 时直接以选项的形式指定要执行的脚本，脚本中可以含有任意数量的 SQL 和 psql 语句，然后 psql 会自动执行此脚本的内容，期间无需与用户进行交互，这就是“非交互模式”的本意。除执行脚本外，非交互模式还支持直接执行一条或多条 SQL 语句，不过语句两侧需要加上双引号。可以看出该模式特别适用于需要执行自动化任务的场景。只需要将任务内容写入脚本文件，然后用某种定时任务工具来设置好定时执行即可。定时任务工具可以采用 pgAgent（其用法在 4.5 节中有详细介绍），或者在 Linux/Unix 下可以使用 `crontab`，在 Windows 下可以使用定时任务规划器。如果某个任务包含很多操作，并且操作必须按顺序执行或者反复执行，那么最好是写成脚本并用 psql 定时执行。由于非交互模式下绝大多数操作逻辑都由脚本实现，因此命令行需要提供的选项很有限。要在非交互模式下执行脚本文件，只需使用 `-f` 选项即可：

```
psql -f some_script_file
```

要在非交互模式下执行 SQL 语句，只需使用 `-c` 选项即可，如果要一次执行多个语句，语句之间请用分号分隔：

```
psql -d postgresql_book -c "DROP TABLE IF EXISTS dross; CREATE SCHEMA staging;"
```

如需了解非交互模式下所支持的全部选项，请参见本书附录 B 中 B.5 节的介绍。

请注意：在脚本中可以使用交互命令。假设你创建了示例 3-1 中的脚本 build_stage.psql。

示例 3-1：带交互式命令的脚本

```
\a \t ❶
SELECT 'CREATE TABLE
 staging.count_to_50 (array_to_string(array_agg('x' || i::text ' varchar(10)));' As
create_sql ❷
FROM generate_series(1,9) As i;
\g create_script.sql ❸
\i create_script.sql ❹
```

❶ 我们希望该脚本执行后输出的结果是能直接运行的 SQL 语句，因此需要用 `\t` 来忽略标题栏的输出，同时使用 `\a` 关闭对齐模式以防止 psql 为对齐输出结果而自动加上换行符。

❷ 生成带 9 个 varchar 列的表。

❸ 使用 `\g` 让所有查询内容都输出到指定文件。

❹ 使用 `\i` 来执行指定的脚本文件。`\i` 的效果等同于非交互模式下的 `-f` 选项。

要执行上述的示例 3-1，在操作系统命令行界面下输入以下命令行即可。

```
psql -f build_stage.psql -d postgresql_book
```

示例 3-1 中借鉴了“如何创建一个有 N 个列的表”这篇博文（http://www.postgresonline.com/journal/archives/230-How-to-create-an-n-column-table-really-fast.html）中所介绍的方法来建表。这篇文章中介绍的方法无需借助中间文件，而是采用了从 PostgreSQL 9.0 版开始支持的 `DO` 命令来执行自动化建表。

3.3 定制psql操作环境

如果你的工作需要整天与 psql 打交道，那么可以对 psql 操作环境进行定制以使其更好地符合自己的使用习惯。psql 在启动阶段会搜索一个名为 psqlrc 的配置文件，如果找到则会顺序执行其中的配置动作，这些配置决定了 psql 的一些行为模式。

在 Linux/Unix 环境中，该文件一般会被命名为 .psqlrc 并放置在 postgres 用户的 home 目录下。在 Windows 上，该文件叫作 psqlrc.conf 并被放置于 %APPDATA%\postgresql 文件夹下，一般来说就是 c:\Users\username\AppData\Roaming\postgresql 文件夹。PostgreSQL 安装完成后找不到此文件是正常的，因为该文件一般需手动创建。该文件中的设置项会覆盖 psql 的默认值。如需了解更多关于此配置文件的信息，请参见官方手册中“psql 参考手册”（http://www.postgresql.org/docs/current/interactive/app-psql.html）这一节的内容。

示例 3-2 中展示了 psqlrc 文件的内容，你可以在其中添加任何 psql 命令以在启动时执行。

示例 3-2：sqlrc 文件内容

```
\pset null 'NULL'
\encoding latin1
\set PROMPT1 '%n@%M:%>%x %/# '
\pset pager always
\timing on
\set qstats92 'SELECT usename, datname, left(query,100) || ''...'' As query
FROM pg_stat_activity WHERE state != ''idle'' ;'
```



psqlrc 文件中 set 命令后跟的操作内容不允许分为多行书写。例如，上面的 qstats92 后跟着的 SQL 语句必须落在同一行中。上文显示为两行仅仅是因为该语句较长导致印刷时放不下，所以必须分成两行。

启动 psql 时，你可以看到该文件中内容被执行后的输出结果。

```
Null display is "NULL".
Timing is on.
Pager is always used.
psql (9.3.2)
Type "help" for help.
postgres@localhost:5442 postgresql_book#
```

有的 psql 设置命令仅适用于 Linux/Unix 环境而不适用于 Windows 环境，反之亦然。但不管在什么操作系统环境下，在指定路径时都应使用 Linux/Unix 风格的正斜杠（/），其目的是为了区别于指定选项时所用的反斜杠（\）。如果希望 psql 启动时跳过加载 psqlrc 文件这一步骤，请加 -X 选项。

如果要删除某个 psql 配置变量或者希望将其设置回默认值，可以使用 \unset 命令，后跟变量名称，比如：\unset qstat92。

下文中我们将介绍一些最常用的 psql 配置项。请注意：这些配置项并不一定要放入 psqlrc 文件中进行管理，它们可以随时按需动态修改。以下两篇博文中介绍了更多关于 psqlrc 文件的用法，可供你参考："DBA 的便捷工具——psqlrc 配置文件"（http://raghavt.blogspot.com.br/2011/11/psqlrc-file-for-dbas.html）和"消除 .psqlrc 文件的执行反馈信息"（http://www.depesz.com/2008/05/18/silencing-commands-in-psqlrc/）。

3.3.1 自定义psql界面提示符

如果你工作时需要使用 psql 连到多台不同的数据库服务器或者多个不同的 database，那么一定会经常用到 \connect 命令。在默认情况下，连到不同数据库以后的命令提示符都是一样的，这很可能会导致误操作的发生。定制后的 psql 界面提示符可以告诉你当前连接的是哪台服务器上的哪个 database，从而避免误操作的发生。在前文提供的示例 psqlrc 文件中，

我们把提示符设成了这样：

```
\set PROMPT1 '%n@%M:%>%x %/# '
```

其中包括了几个元素：登录角色（`%n`），主机名（`%M`），侦听端口（`%>`），事务状态（`%x`）以及当前使用的 database 名（`%/`）。这种写法可能有点详细得过了头，你可以按需裁剪一下。提示符所支持的完整符号列表可从官方手册的“psql 参考手册”（http://www.postgresql.org/docs/current/interactive/app-psql.html）这一节中找到。

连接到数据库后，提示符看起来会像下面这样。

```
postgres@localhost:5442 postgresql_book#
```

执行 `\connect postgis_book` 切换目标数据库后，提示符会变成下面这样。

```
postgres@localhost:5442 postgis_book#
```

3.3.2　语句执行时间统计

有时候我们可能会需要 psql 打印出执行每个语句所消耗的时间。通过 `\timing` 命令来打开或者关闭执行时间统计开关。

打开执行时间统计开关时，每个查询执行完毕的输出结果中都会附带执行时长。例如：使用 `\timing on` 命令执行 `SELECT COUNT(*) FROM pg_tables` 后，输出如下：

```
count
--------
73
(1 row)
Time: 18.650 ms
```

3.3.3　事务自动提交

默认情况下，`AUTOCOMMIT`（自动提交）是开着的，也就是说任何一个 SQL 语句执行完毕后，它所做的数据修改都会被立即提交，这种情况下每个语句都是一个独立的事务，一旦执行完毕后其结果就不可撤销。如果你需要运行大量的 DML 语句并且这些语句还未经充分测试，那么自动提交功能会带来大麻烦，此时有必要关闭事务自动提交机制来对数据进行保护。

请先关闭自动提交功能：`\set AUTOCOMMIT off`；然后就可以按需对事务进行回滚了：

```
UPDATE census.facts SET short_name = 'This is a mistake.';
```

要回滚事务，请执行：

```
ROLLBACK;
```

要提交事务，请执行：

```
COMMIT;
```



在非自动提交模式下请一定要记得在最后提交事务；否则未提交的事务会在退出 psql 时自动回滚掉。

3.3.4 命令别名

你可以使用 \set 来为某个命令创建别名，我们建议你将全局性的别名写到 psqlrc 文件中去。例如，如果你每十分钟就要执行一次 EXPLAIN ANALYZE VERBOSE，那么每次完全重输一遍会很烦人，可以为它起一个别名。

```
\set eav 'EXPLAIN ANALYZE VERBOSE'
```

这样在任何需要用到 EXPLAIN ANALYZE VERBOSE 命令的地方，都可键入 :eav 替代（前面的冒号表示这是一个需要展开的命令变量）。

```
:eav SELECT COUNT(*) FROM pg_tables;
```

你甚至可以把常用查询做成别名存入 psqlrc 文件中，就类似前面出现过的 qstats91 和 qstat92 一样。我们建议别名使用小写字母，以避免与 psql 环境变量冲突，环境变量都是大写的。

3.3.5 取出前面执行过的命令行

跟许多命令行工具一样，你在 psql 中也可以用向上方向键快速找出之前执行过的历史命令。HISTSIZE 环境变量决定了系统存储的历史命令行的数量。例如：\set HISTSIZE 10 会将可追溯的历史命令数量设定为最多 10 条。

如果你正在编写一个非常复杂的查询语句或者执行一系列很关键的更新操作，那么可能会需要将这些语句存入指定的文件中以备后查，可以使用 HISTFILE 环境变量来实现此功能。

```
\set HISTFILE ~/.psql_history- :HOST - :DBNAME
```



Windows 环境下不支持保存历史命令，除非是使用 Cygwin 来模拟 Unix 环境。

3.4 psql使用技巧

本节中，我们将向你介绍一些被湮没在大量的 psql 文档中的特殊技巧。

3.4.1 执行shell命令

psql 中通过 `\!` 可以直接执行操作系统命令。比如你使用的是 Windows 环境，需要列出当前工作目录的内容，那么不需要退出 psql 环境，只需要直接在 psql 界面上执行 `\! dir` 即可。

3.4.2 用watch命令重复执行语句

`\watch` 命令是 PostgreSQL 9.3 版中为 psql 引入的一项新功能。它可以实现以固定的频率反复执行某个语句，以便持续观察其输出。例如你需要持续监控系统中当前正在执行的所有语句的情况，那么可以执行示例 3-3 中的操作。

示例 3-3：每 10 秒钟查询一次所有数据库连接上的活跃负载

```
SELECT datname, waiting, query
FROM pg_stat_activity
WHERE state = 'active' AND pid != pg_backend_pid(); \watch 10
```

邮
电

虽然设计 `\watch` 命令的本意是用于反复执行监控类查询语句以便于持续观察系统状态，但你也可以将其用于重复执行其他指定的语句，`watch` 命令并不关心语句本身的内容是什么。比如，示例 3-4 实现了每 5 秒记录一次系统负载信息。

示例 3-4：每 5 秒记录一次系统负载情况

```
SELECT \* INTO log_activity FROM pg_stat_activity; ❶
           INSERT INTO log_activity SELECT * FROM pg_stat_activity; \watch 5 ❷
```

❶ 以 pg_stat_activity 为模板新建一张结构和数据都完全相同的 log_activity 表。

❷ 每 5 秒重复一次将 pg_stat_activity 最新数据导入 log_activity 的动作。

如果需要终止 `watch` 进程，请执行 `CTRL-X` 加 `CTRL-C`。很显然，watch 动作一定是只能在 psql 的交互模式下使用。

你可以在 Michael Paquier 的博文“psql 中 watch 功能的使用”（http://michael.otacoo.com/postgresql-2/postgres-9-3-feature-highlight-watch-in-psql/）中找到更多关于 watch 用法的例子。

3.4.3 显示对象信息

有多条 psql 命令都能用于显示数据库对象列表，并附带给出每个对象的详细信息。示例 3-5 展示了如何列出 `pg_catalog` 中以 `pg_t` 打头的所有表的信息，同时附带这些表所占空间的大小。

示例 3-5：使用 `\dt+` 命令列出表信息

```
\dt+ pg_catalog.pg_t*

Schema     |       Name       | Type  |  Owner   |  Size  | Description
-----------+------------------+-------+----------+--------+-------------
pg_catalog | pg_tablespace    | table | postgres | 40 kB  |
pg_catalog | pg_trigger       | table | postgres | 16 kB  |
pg_catalog | pg_ts_config     | table | postgres | 40 kB  |
pg_catalog | pg_ts_config_map | table | postgres | 48 kB  |
pg_catalog | pg_ts_dict       | table | postgres | 40 kB  |
pg_catalog | pg_ts_parser     | table | postgres | 40 kB  |
pg_catalog | pg_ts_template   | table | postgres | 40 kB  |
pg_catalog | pg_type          | table | postgres | 112 kB |
```

如果需要查询某个特定对象的详细信息，可以使用 `\d+` 命令。示例 3-6 演示了如何查出 `pg_ts_dict` 表的详细信息：

示例 3-6：通过 `\d+` 命令得到对象的详细信息

```
\d+ pg_ts_dict

表 "pg_catalog.pg_ts_dict"
Column         | Type | Modifiers | Storage  | Stats target | Description
---------------+------+-----------+----------+--------------+-------------
dictname       | name | not null  | plain    |              |
dictnamespace  | oid  | not null  | plain    |              |
dictowner      | oid  | not null  | plain    |              |
dicttemplate   | oid  | not null  | plain    |              |
dictinitoption | text |           | extended |              |
Indexes:
"pg_ts_dict_dictname_index" UNIQUE, btree (dictname, dictnamespace)
"pg_ts_dict_oid_index" UNIQUE, btree (oid)
Has OIDs: yes
```

3.5　使用psql实现数据的导入和导出

psql 支持一个叫作 `\copy` 的命令，该命令可以将数据导出到文本文件中，同时也可以从文本文件中导入数据。文本文件中默认使用制表符作为分隔符，当然你也可以指定使用其他分隔符。文本中必须使用换行符来分隔不同的行，否则无法正确区分两行记录。我们采用美国人口普查数据网站（http://factfinder2.census.gov/）上提供的马萨诸塞州人口统计数据来作为我们的第一个例子。你可以从链接 http://www.postgresonline.com/downloads/postgresql_book_2e.zip 中下载我们接下来要用到的 DEC_10_SF1_QTH1_with_ann.csv 文件。

3.5.1　使用psql进行数据导入

需要导入非规范化数据或者需要导入我们不太了解其特征的数据时，我们一般建议这么做：首先，新建一个独立的 schema 来作为数据过渡区，然后将新数据导入此 schema 中；

然后，通过一些查询来摸清这些数据的特性；最后才把这些数据分门别类导入到正式的产品表中并删除之前建立的过渡区 schema。

在将数据导入 PostgreSQL 之前，需要先建立一张表来容纳新数据，而且表的列数和数据类型必须与待导入的数据一致。如果一个待导入的文本文件的内部列和记录结构清晰有序，那么这个建表步骤理论上是可以省略的，因为 psql 可以自动判断每个列的类型并自动把表建好，但为了避免出现 psql 自动判断数据类型出错的情况，我们认为这个步骤最好不要省略。psql 把整个导入过程当成一个完整的事务来处理，因此如果导入数据时遇到任何错误，那么整个导入动作所做的修改会完全回滚掉。如果你对源文件中数据的特点并不完全了解，我们建议你用最宽松的条件来建容纳表，等数据导入之后再对数据进行细化加工。例如，如果你不确定某一列是否全都是数字，那么可以将该列的数据类型设置为 `character varying`，等数据导入之后再执行检查，然后稍后再进行转换。

打开 psql 并执行示例 3-7 中的命令。

示例 3-7：使用 psql 导入数据

```
\connect postgresql_book
\cd /postgresql_book/ch03
\copy staging.factfinder_import FROM DEC_10_SF1_QTH1_with_ann.csv CSV
```

上面的示例中，我们在 psql 的交互模式下执行了导入过程：首先连接数据库，然后使用 `\cd` 切换到含有数据源文件的目录，最后再执行 `\copy` 导入动作。因为 `\copy` 命令支持的默认分隔符是制表符，所以我们必须在命令行中额外指明源文件是用逗号作为分隔符的 `CSV` 格式。

如果源文件使用了一些非标准的分隔符，比如竖杠“|”，那么也请在命令中指明：

```
\copy sometable FROM somefile.txt DELIMITER '|';
```

如果希望把文本中的空值替换为别的内容再导入，可以用 `NULL AS` 来标记要替换的内容：

```
\copy sometable FROM somefile.txt NULL As '';
```



请不要将 psql 中的 `\copy` 命令与 SQL 语言提供的 `COPY` 语句相混淆。psql 是一个客户端工具，所有路径都是相对于已连接客户端进行解释的。而 SQL copy 是基于服务器的，并在 `postgrese` 服务操作系统账户的环境下运行。因此输入文件必须驻留在可由 `postgres` 服务账户访问的路径中。我们在“使用 psql 导入固定宽度数据”这篇博文（http://www.postgresonline.com/journal/archives/157-Import-fixed-width-data-into-PostgreSQL-with-just-psql.html）中详细介绍了二者的区别。

3.5.2 使用psql进行数据导出

psql 的数据导出功能比导入功能更简单、更灵活，你甚至可以指定仅导出某张表的某一部分记录。导出时使用的依然是 psql 的 `\copy` 命令。示例 3-8 中，我们将演示如何将前面刚刚导入库中的数据再导回（以制表符为分隔符）到文本中。

示例 3-8：使用 psql 导出数据

```
\connect postgresql_book
\copy (SELECT * FROM staging.factfinder_import WHERE s01 ~ E'^[0-9]+' ) TO '/
test.tab'
WITH DELIMITER E'\t' CSV HEADER
```

默认情况下 psql 导出数据时会使用 tab 键作为分隔符。然而，以这种格式导出时默认不导出标题行，你可以通过指定 `HEADER` 选项来要求导出标题行（详见示例 3-9），请注意该选项仅当输出格式为 `CSV` 时才可使用。

示例 3-9：使用 psql 导出数据

```
\connect postgresql_book
\copy staging.factfinder_import TO '/test.csv' WITH CSV HEADER QUOTE '"' FORCE
QUOTE *
```

`FORCE QUOTE *` 表示输出的所有列的前后都将加上引用符，这个引用符默认就是双引号，但为清晰起见，我们还是显式指定了一下。

3.5.3 从外部程序复制数据以及将数据复制到外部程序

从 PostgreSQL 9.3 版开始，psql 开始支持从命令行程序的输出中取数据并将数据转储到表中，这类命令行程序包括 `curl`、`ls` 和 `wget` 等。示例 3-10 演示了如何使用 `dir` 命令将一个目录下的文件列表导入表中。

示例 3-10：使用 psql 导入某个目录下的文件列表

```
\connect postgresql_book
CREATE TABLE dir_list (filename text);
\copy dir_list FROM PROGRAM 'dir C:\projects /b'
```

Hubert Lubaczewski 在他的博文“以基于管道的方式将数据复制到外部程序或者从外部程序复制数据”（http://www.depesz.com/2013/02/28/waiting-for-9-3-add-support-for-piping-copy-tofrom-an-external-program/）中介绍了更多关于使用 `\copy` 命令的例子。

3.6 使用psql制作简单的报表

你也许会觉得难以置信，但 psql 的确能够制作简单的 HTML 报表。请执行以下命令并查看输出的 HTML 报表，应该是类似图 3-1 所展示的样子。

```
psql -d postgresql_book -H -c
"SELECT category, count(*) As num_per_cat
FROM pg_settings
WHERE category LIKE '%Query%'
GROUP BY category
ORDER BY category;" -o test.html
```

category	num_per_cat
Query Tuning / Genetic Query Optimizer	7
Query Tuning / Other Planner Options	5
Query Tuning / Planner Cost Constants	6
Query Tuning / Planner Method Configuration	11
Statistics / Query and Index Statistics Collector	6

(5 rows)

图 3-1：简单 HTML 报表

上面的报表看起来还算凑合，但这仅仅是输出了一个 HTML 表格，还称不上是一个完全符合格式要求的 HTML 文档。为了让本报表的内容更丰富一些，我们需要写一个脚本来作为辅助，内容见示例 3-11。

示例 3-11：编写 settings_report.psql 文件来设置报表内容

```
\o settings_report.html ❶
\T 'cellspacing=0 cellpadding=0' ❷
\qecho '<html><head><style>H2{color:maroon}</style>' ❸
\qecho '<title>PostgreSQL Settings</title></head><body>'
\qecho '<table><tr valign=''top''><td><h2>Planner Settings</h2>'
\x on ❹
\t on ❺
\pset format html ❻
SELECT category, string_agg(name || '=' || setting, E'\n' ORDER BY name) As set-
tings ❼
FROM pg_settings
WHERE category LIKE '%Planner%'
GROUP BY category
ORDER BY category;
\H
\qecho '</td><td><h2>File Locations</h2>'
\x off ❽
\t on
\pset format html
SELECT name, setting FROM pg_settings WHERE category = 'File Locations' ORDER BY
name;
\qecho '<h2>Memory Settings</h2>'
SELECT name, setting, unit FROM pg_settings WHERE category ILIKE '%memory%' ORDER
BY name;
\qecho '</td></tr></table>'
\qecho '</body></html>'
\o
```

❶ 指定查询结果输出到一个文件中。

❷ HTML 表格的输出格式设置。

❸ 添加一些附加的 HTML 代码。

❹ 打开记录输出的展开模式。即重复每一个记录的列标题，并将每一个记录的每一列作为一个单独的记录输出。

❺ 设置“是否仅输出记录”开关。如果此开关是打开的，则会忽略列标题和行计数。

❻ 指定按 HTML 表格格式输出结果。

❼ `string_agg()` 是 PostgreSQL 9.0 版引入的一个函数，可以将聚合运算中被划为同组的字符串值合并为单个字符串。

❽ 关闭记录输出的展开模式，这样第二个和第三个查询结果在报表上的输出格式应该是每条记录仅占一行。

示例 3-11 演示了通过灵活使用 SQL 和 psql 命令可以创建出一个内容丰富的综合性分层报表。要运行上面示例中的脚本有两个方法：可以使用 psql 以交互方式连接并执行 `\i settings_report.psql`，也可以在操作系统的命令行界面上运行 `psql -f settings_report.psql`。settings_report.html 生成的输出结果如图 3-2 所示。

Planner Settings

category	Query Tuning / Other Planner Options
settings	constraint_exclusion=partition cursor_tuple_fraction=0.1 default_statistics_target=100 from_collapse_limit=8 join_collapse_limit=8
category	Query Tuning / Planner Cost Constants
settings	cpu_index_tuple_cost=0.005 cpu_operator_cost=0.0025 cpu_tuple_cost=0.01 effective_cache_size=16384 random_page_cost=4 seq_page_cost=1
category	Query Tuning / Planner Method Configuration
settings	enable_bitmapscan=on enable_hashagg=on enable_hashjoin=on enable_indexonlyscan=on enable_indexscan=on

File Locations

config_file	C:/projects/pg/pg92edb/data/postgresql.conf
data_directory	C:/projects/pg/pg92edb/data
external_pid_file	
hba_file	C:/projects/pg/pg92edb/data/pg_hba.conf
ident_file	C:/projects/pg/pg92edb/data/pg_ident.conf

Memory Settings

maintenance_work_mem	16384	kB
max_prepared_transactions	0	
max_stack_depth	2048	kB
shared_buffers	4096	8kB
temp_buffers	1024	8kB
track_activity_query_size	1024	
work_mem	1024	kB

图 3-2：复杂 HTML 报表

通过以上脚本，可以实现将多个查询的输出结果整合到一个报表中，如果要定时生成此报表，请使用 `pgAgent` 或者 `crontab`。

第4章

pgAdmin的使用

pgAdmin 是一款 PostgreSQL 图形化管理工具，其可靠性和实用性已久经验证，目前已发展到第三个大版本，一般称为 pgAdmin III。虽然该工具有其缺点，但开发组对 bug 的修复一直都很及时，而且也在不停地为其添加新的功能特性。因为其定位是 PostgreSQL 官方的图形化管理工具，并且在很多 PostgreSQL 发行包中都会附带，所以它会一直与最新版本的 PostgreSQL 保持同步更新。如果新版本的 PostgreSQL 中引入了一个新特性，那么最新版本的 pgAdmin 中一定会支持对此特性进行管理。如果你是 PostgreSQL 新手，那么选择 pgAdmin 作为入门工具肯定没错。

4.1 pgAdmin入门

你可以从 pgAdmin 的官网 www.pgadmin.org 下载 pgAdmin 的安装包，网站上还提供了 pgAdmin 的使用手册，你可以仔细研读一下。该工具的功能设计清晰有序，而且大部分功能是自解释的，也就是说你仅凭摸索而无需专门指导即可了解其使用方法。对于那些爱“尝鲜”的人们来说，可以尝试使用测试版，PostgreSQL 社区将很感谢你对测试版进行试用并反馈 bug。

4.1.1 功能概览

下面会先列出我们认为最精华的部分功能，不过这只是小试牛刀，更多内容请参见 pgAdmin 官网上的功能介绍页面（http://pgadmin.org/features.php）。

- 执行计划的图形化解释功能

  该功能能够以图形化方式展示规划器的内部工作过程，有了该功能以后就不再需要在冗长难懂的执行计划文本中艰难跋涉寻求真相了。

- SQL执行面板

  不管在界面上执行什么样的操作，pgAdmin 最终也是要通过 SQL 语句来与 PostgreSQL 服务端进行交互，系统允许你查看这种底层 SQL。当你使用图形化界面对数据库服务器进行操作时，pgAdmin 会自动在 SQL 窗格中展示这些自动生成的 SQL 语句。对于新手来说，研究这种自动生成的 SQL 是极好的学习途径。对专家来说，好好利用这种自动生成的 SQL 可以节省大量时间。

- postgresql.conf和pg_hba.conf等配置文件的图形化编辑器

  你不再需要四处寻找配置文件的位置并使用文本编辑器来修改它们，在 pgAdmin 上可以一站式搞定。

- 数据导入和导出

  pgAdmin 能够轻易地将语句查询结果导出为 CSV 文件或者基于其他分隔符的文本文件，当然也可以将这类文件导入到数据库中。它甚至支持将表数据导出为 HTML 格式，因此可以当作一个简易的一键式报表服务器来用。

- 备份与恢复向导

  如果你记不住 `pg_restore` 和 `pg_dump` 命令的大量选项，那没关系，pgAdmin 提供了一个很友好的图形化界面来帮你设定这些选项，通过调整这些选项可以实现对 database、schema、单张表以及全局对象进行定制化的备份或者恢复。你可以在备份恢复界面的“消息”选项卡上看到系统自动生成的 `pg_dump` 或 `pg_restore` 命令行语句，如果你觉得有必要，完全可以把这些语句复制下来作为例句使用。

- 授权向导

  该功能可以帮助你一次性对很多数据库对象进行授权或者解除授权操作，从而大大节省你的时间。

- pgScript脚本执行引擎

  pgScript 是一种速度很快但不太“正规”的脚本执行机制，该机制不要求整个脚本中执行的所有操作构成一个事务。在 pgScript 中，你可以在一个循环语句的每一次循环中都执行一次提交操作，如果是在函数中，那么只能是所有循环都结束了最后才能执行提交。很遗憾的是，这种灵活的机制只能在 pgAdmin 中使用。

- 插件架构

  该架构让用户只需点一下鼠标即可加载一个新的 pgAdmin 插件。你甚至可以通过此机制来加入你自己开发的插件。我们在"pgAdmin III 1.1.3 版中插件机制的变化"这篇博文（http://www.postgresonline.com/journal/archives/180-pgAdmin113plugins_postgis.html）中讨论了该特性。

- pgAgent定时任务工具

  pgAgent 是一种跨平台的定时任务计划工具，我们后续将使用一整节的篇幅来介绍它。pgAdmin 提供了一套很完善的用于访问 pgAgent 的接口。

4.1.2 如何连接到PostgreSQL服务器

通过 pgAdmin 连接到 PostgreSQL 服务器是很简单的，其属性页和高级功能页面如图 4-1 所示。

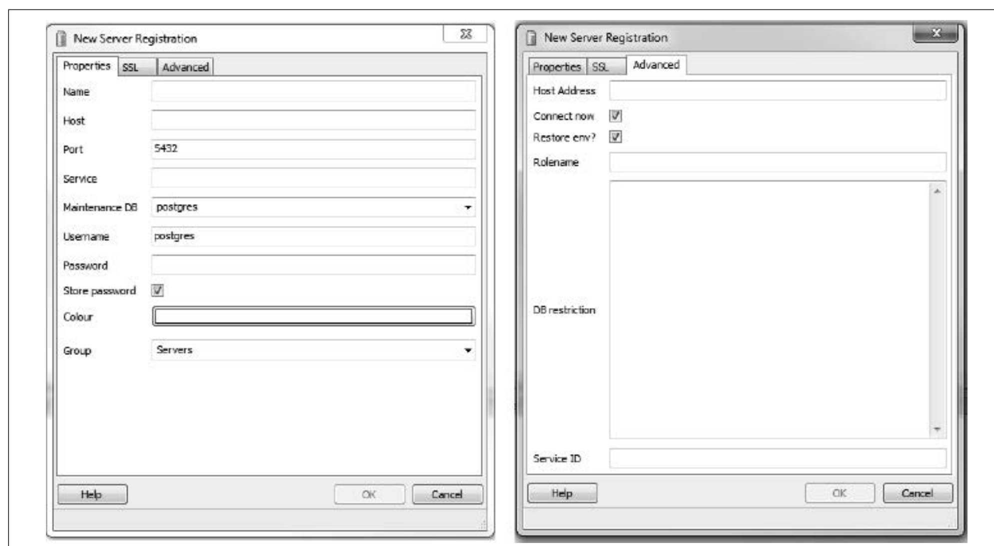


图 4-1：pgAdmin 注册服务器连接对话框

4.1.3 pgAdmin界面导航

pgAdmin 界面左侧的树状目录布局看起来很直观，其中展示了所有的数据库对象。你可以进入 Options（选项）选项卡中，勾选掉你不希望看到的数据库对象类型，这样左侧的目录树就会精简很多。可以通过菜单栏上的 Tools（工具）→ Options（选项）→ Browser（浏览器）来打开目录树定制面板，你将看到如图 4-2 的界面。

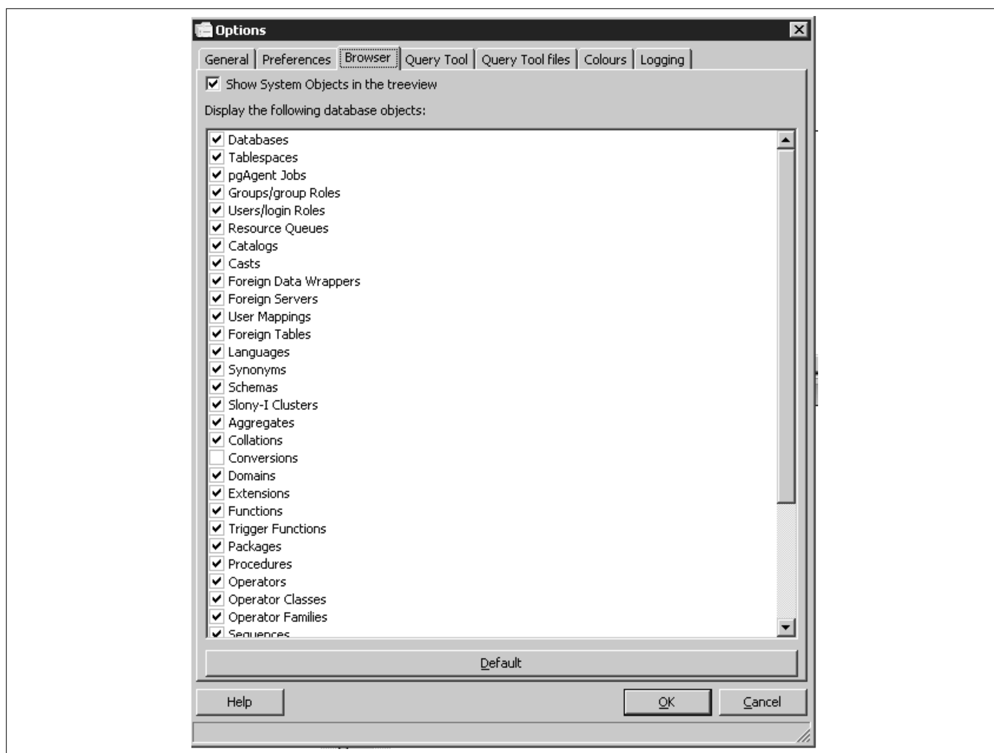


图 4-2：在 pgAdmin 的树状浏览目录中隐藏或者显示特定类型的数据库对象

如果在界面上勾选了“在树状目录中显示系统对象”，那么你将看到 PostgreSQL 服务器的内部对象，包括内部函数、系统表、表的隐藏列等。你还将看到 PostgreSQL 系统 schema 中存储的元数据，包括 `information_schema` 和 `pg_catalog` 这两个 catalog 中的内容。其中 `information_schema` 是 ANSI SQL 标准中规定必须要有的，因此在别的数据库（比如 MySQL 和 SQL Server）中也会存在。你可能会认出一些在使用其他数据库产品时曾经见过的表和列。



pgAdmin 左侧的目录树并不总是和数据库服务端实时保持一致的。例如，如果有人通过一个会话连接到服务端并修改了某表的结构，此时另一个人通过另一个会话打开的 pgAdmin 目录视图上的表结构并不会实时更新。在最新的版本中有一个设置，可用于强制自动刷新，但请理解无论如何都会有一定的时延。

4.2 pgAdmin功能特性介绍

pgAdmin 工具的功能丰富而强大，本书的篇幅不足以完全描述，因此我们仅重点介绍一些

比较常用的功能。

4.2.1 在pgAdmin中调用psql

尽管 pgAdmin 拥有功能强大的图形界面，但有些时候还是离不开 psql。比如需要执行 pg_dump 或其他数据转储工具生成的体积庞大的 SQL 文件时，psql 就更合适。从 pgAdmin 界面上可以很容易地打开 psql 工具，只需点击“插件”菜单下的“PSQL Console”项即可，如图 4-3 所示。这样就会启动一个 psql 窗口并连接到当前 pgAdmin 环境中已连接的 database 上，然后你可以使用 `\cd` 以及 `\i` 命令来改变目录并执行 SQL 文件。

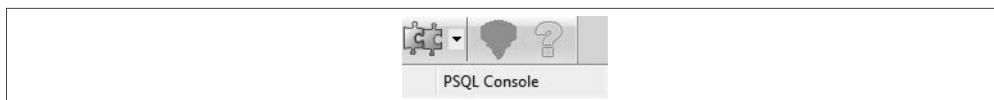


图 4-3：psql 插件

请注意：该功能需要确保针对某 database 的连接已建立好，因此只有当 pgAdmin 已连上 PostgreSQL 服务器并选中某个 database 时，“插件”菜单下的“PSQL Console”项才会变成可用状态。

4.2.2 在pgAdmin中编辑postgresql.conf和pg_hba.conf文件

只要服务器上安装了 adminpack 扩展包，你就可以在 pgAdmin 界面上直接编辑配置文件。一般来说，PostgreSQL 的一键式安装包都会自动安装好 adminpack 扩展包，你可以看到 Server Configuration（服务器配置）菜单已启用，如图 4-4 所示。



图 4-4：pgAdmin 配置文件编辑器

如果你的 pgAdmin 已连接到 PostgreSQL 服务器但 Server Configuration（服务器配置）菜单却是灰的，那么要么是没安装 adminpack，要么是你不是以超级用户身份登录的。如果要在 PostgreSQL 9.0 或者更早版本的服务器上安装 adminpack，请以 `postgres` 超级用户身份登录然后运行脚本 share/contrib/adminpack.sql。对于 PostgreSQL 9.1 以及之后的版本来说，请以 `postgres` 用户身份登录并执行 `CREATE EXTENSION adminpack`，或者也可以通过图形界面来安装，如图 4-5 所示。安装好以后请断开与服务器之间的连接并重连，然后就可以看到菜单已可点击。

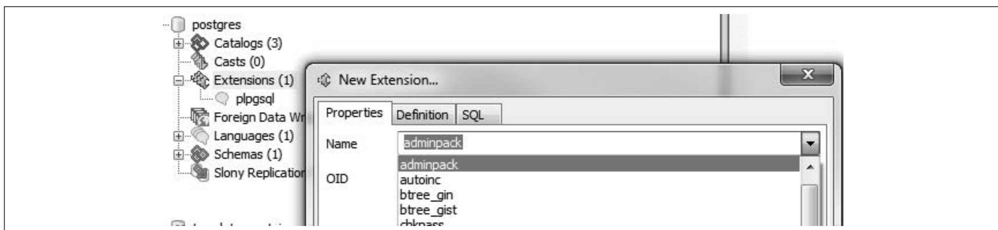


图 4-5：使用 pgAdmin 安装扩展包

4.2.3　创建数据库资产并设置权限

pgAdmin 允许你创建各种数据库资产并对其进行权限设置。

1. 创建数据库以及其他数据库资产

利用 pgAdmin 创建一个新的数据库是非常简单的，只需右键单击树上的 database 节点并选择 New Database（新建数据库）即可，如图 4-6 所示。Definition（定义）选项卡上提供了一个下拉菜单供选择建库所用的模板数据库，我们在 2.4.1 节中介绍过相关内容。

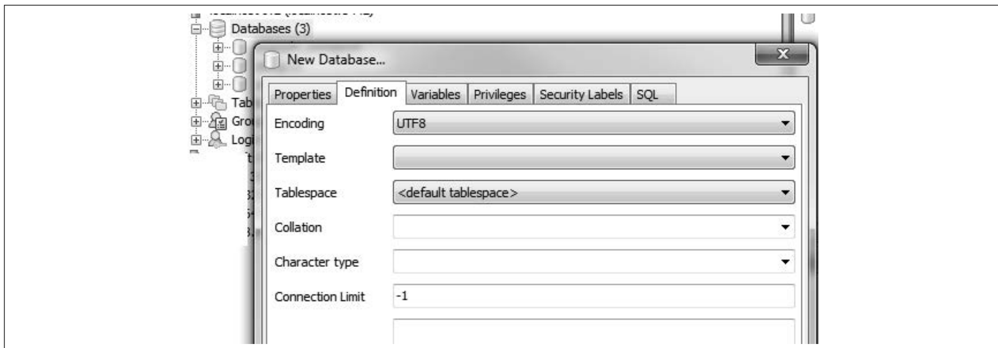


图 4-6：创建新数据库

创建角色、schema 和其他数据库资产的步骤是类似的，都有一些对应的相关页面供你设置其他属性。

2. 权限管理

在 PostgreSQL 数据库资产权限管理方面，不会有比 pgAdmin 的授权向导更好的管理工具了，你可以通过菜单栏上的 Tools（工具）→ Grant Wizard（授权向导）打开其页面。如同其他许多功能项一样，在成功连到数据库之前，其菜单项一直都是灰的。另外该菜单项对于当前树状目录上的焦点位置也很敏感，点击到不同位置时，该菜单项就会显示出可用或者不可用等不同状态。例如，要为 `cesus` 这个 schema 中的项设置权限，请在目录树上选中此 schema 并点开授权向导，界面如图 4-7 所示。然后你就可以选择所有或者部分项，然后切换到“权限”选项卡上以设置你想要授予的角色和权限。

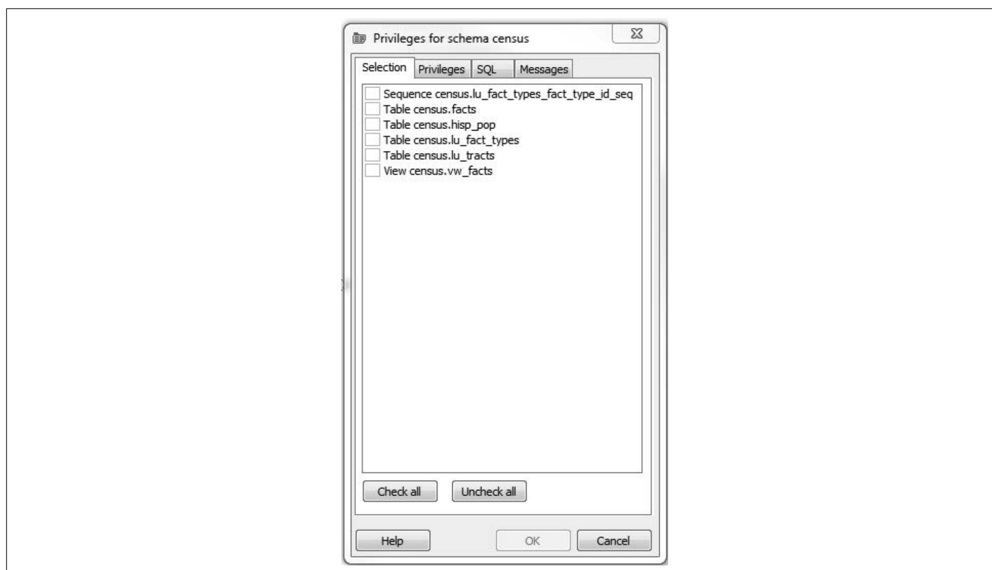


图 4-7：授权向导

除了为已有对象授权以外，我们日常遇到更多的一个场景是为一个 schema 或者 database 中新建的对象设置默认权限。要执行此类授权，请右键单击 schema 或者 database 对象节点，然后选择“属性”菜单项，然后在弹出的界面上点击切换到“默认权限”选项卡，如图 4-8 所示。请注意：“默认权限”这一功能特性仅仅适用于 PostgreSQL 9.0 及之后的版本。

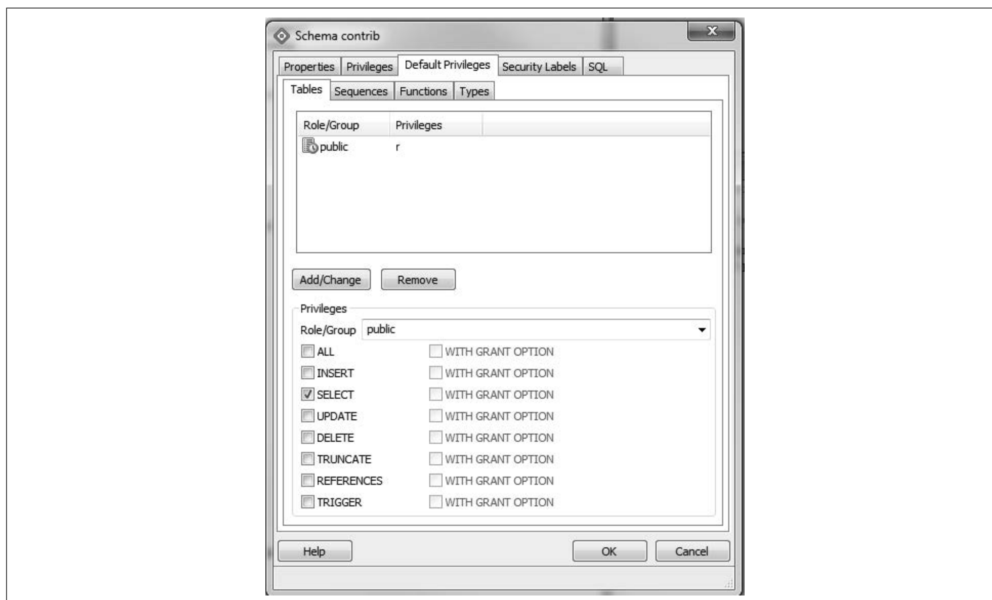


图 4-8：默认授权管理

当为 schema 授予默认权限时，请记得一定要为相应的组角色授予访问此 schema 的权限。

4.2.4　数据导入和导出

同 psql 一样，pgAdmin 也可以导入和导出文本文件。

1. 导入文件

pgAdmin 的导入功能其实是对 psql 的 `\copy` 命令做了一层封装，并要求导入数据的目的表必须已建好。要实现数据导入，请在要导入数据的表上单击鼠标右键，如图 4-9 所示。

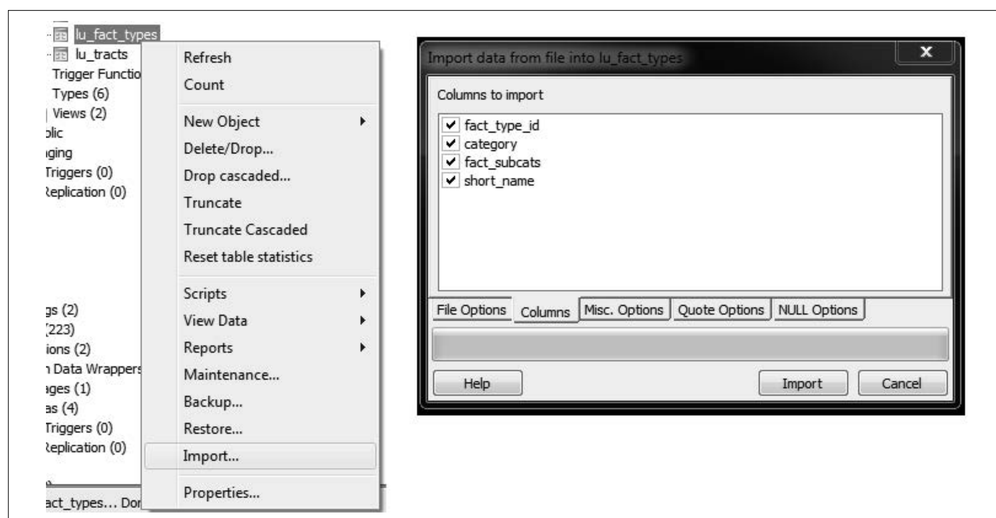


图 4-9：Import（导入）菜单

2. 将数据导出为结构化文件或者报表格式

除了导入数据，你还可以将数据导出为分隔符分隔的文本文件格式以及 HTML 或者 XML 格式。要导出为分隔符文本格式，请按以下步骤操作。

(1) 打开查询窗口（ ）。
(2) 编写查询语句。
(3) 执行查询语句。
(4) 点击菜单栏上的 File（文件）→Export（导出）。
(5) 按照图 4-10 填写设置内容。

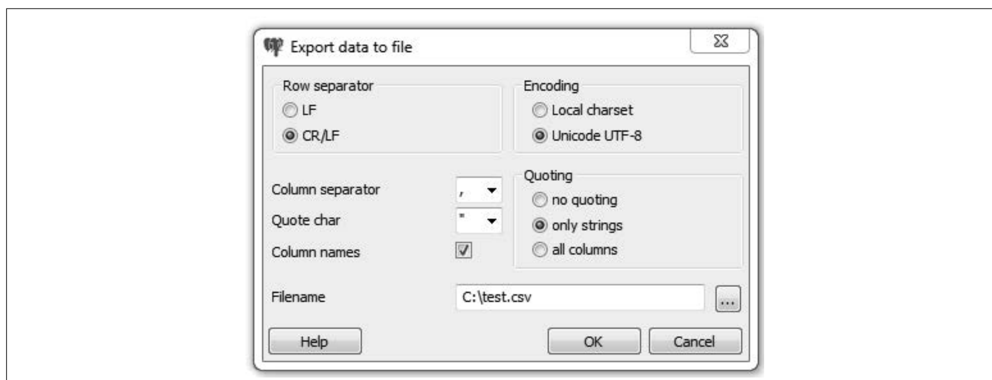


图 4-10：Export（导出）菜单

导出为 HTML 或者 XML 的步骤非常类似，唯一的差别在于需要点击菜单栏上的 File（文件）→Quick Report（快速报表）选项，参见图 4-11。

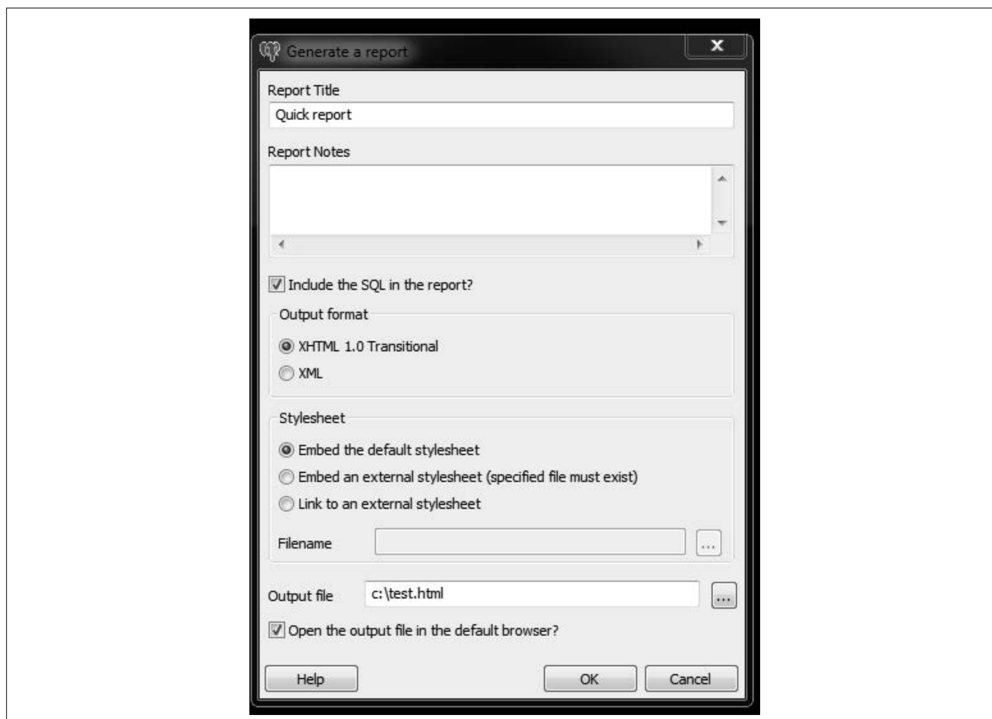


图 4-11：导出报表选项

4.2.5　备份与恢复

pgAdmin 为 `pg_dump` 和 `pg_restore` 提供了图形化的操作界面，相关具体功能已在 2.7 节中

介绍过。本节内容中，我们将重复使用一些前面已经使用过的例子，不过是使用 pgAdmin 来执行操作，而非使用命令行。

如果你的机器上安装了多个版本的 PostgreSQL 或 pgAdmin，我们建议你先确认 pgAdmin 指向了正确版本的 PostgreSQL 的 bin 目录（即 pg_dump、pg_restore 等命令行工具所在目录），可以通过检查 pgAdmin 的 bin 目录设置来确认，如图 4-12 所示。

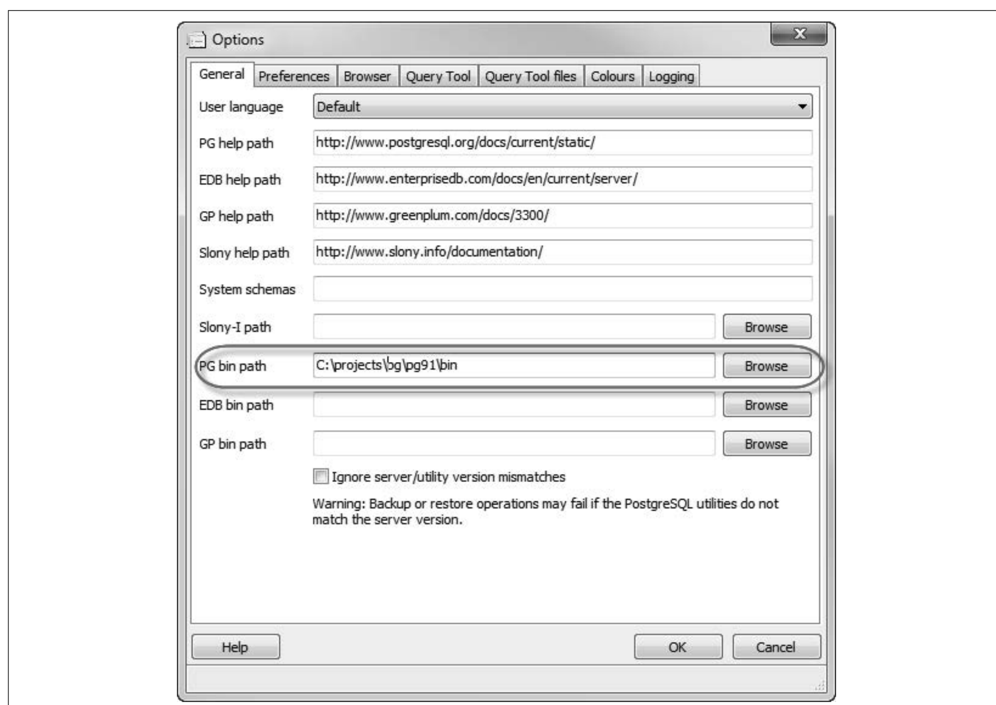


图 4-12：pgAdmin File（文件）→Options（选项）菜单



如果你是在对一台远程服务器进行备份或者恢复操作，或者你操作的数据库数量特别庞大，那么我们建议你使用命令行工具而不要使用 pgAdmin，因为此种情况下操作过程本来就已经很耗时，使用 pgAdmin 操作会使得耗时和复杂度增加。另外也请牢记：对于 pg_dump 转储的自定义压缩格式、TAR 包格式、目录格式这三种二进制格式的备份文件，必须使用与 pg_dump 相同或者更新版本的 pg_restore 工具来进行恢复。

1. 完整备份一个database中的数据

在 2.7.1 节中，我们已经演示了如何完整备份一个 database。以下我们使用 pgAdmin 界面重复演示一遍操作过程，请右键单击待备份的 database，并选择 Custom（自定义）格式，如图 4-13 所示。



图 4-13：备份 database

2. 备份系统级对象

pgAdmin 为 `pg_dumpall` 提供了一个图形化界面，用于对系统对象进行备份。要使用该界面，请先连接到希望备份的 PostgreSQL 服务器。然后从顶部菜单中选择 Tools（工具）→ Backup Globals（全局备份）。

pgAdmin 不支持指定备份哪些全局对象，但在 `pg_dumpall` 的命令行界面上是可以的。pgAdmin 默认会备份所有的系统表空间和角色。

如果你希望备份整个服务器端的所有数据，可以通过点击菜单栏上的 Tools（工具）→ Backup Server（备份服务器）来实现。

3. 选择性地备份部分数据库资产

pgAdmin 为 `pg_dump` 的选择性备份功能提供了一个图形化接口。在希望备份的数据库资产上右键单击，然后在弹出的菜单中选择 Backup（备份）（如图 4-14 所示）。你可以选择备份整个 database、一个特定的 schema 或者任何其他数据库资产。

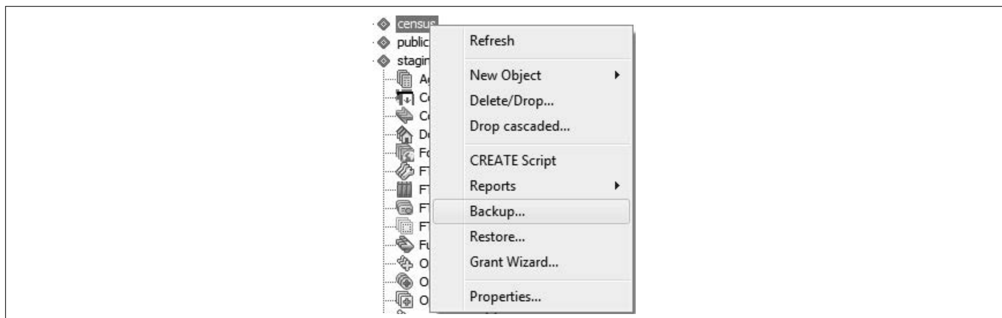


图 4-14：pgAdmin 的 schema 备份

如果希望仅备份当前图形界面上选中的数据库资产，那么你可以忽略备份界面上的其他选项卡（如图 4-13 所示），只用默认设置即可。当然，你也可以切换到 Objects（对象）选项卡选择备份更多对象，如图 4-15 所示。

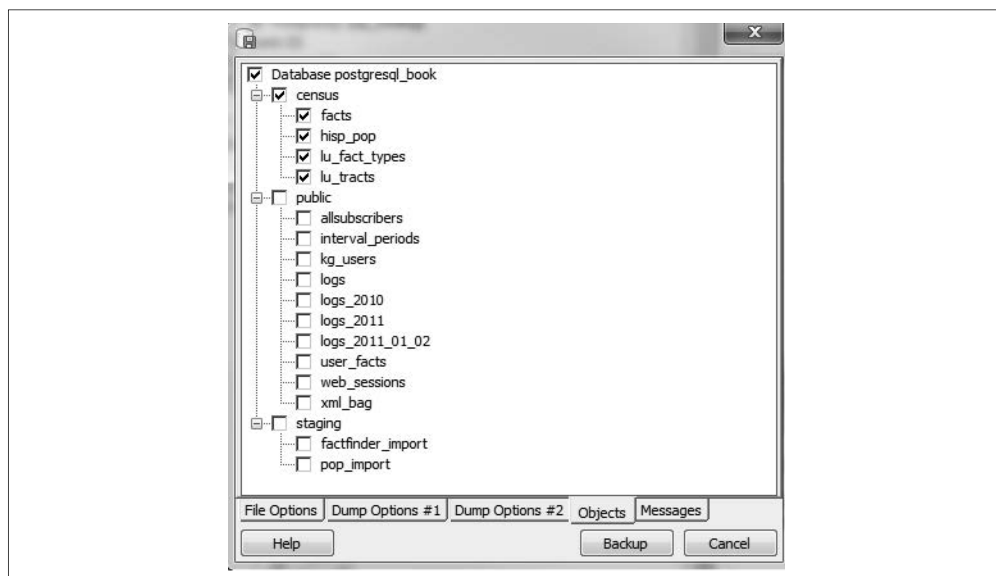


图 4-15：pgAdmin 的选择性备份 Objects（对象）选项卡



pgAdmin 在后台其实就是调用了 `pg_dump` 命令行工具来实施备份动作，如果你希望了解 pgAdmin 最终使用的命令是什么样子，那么可以在点击 Backup（备份）按钮开始执行备份后切换到备份界面最右侧的 Messages（消息）选项卡，其中会记录系统自动生成的带实参的 `pg_dump` 命令行。

4.3 pgScript脚本机制

pgScript 是 pgAdmin 内置的一种脚本机制，非常适合于重复执行 SQL 任务的场景。相比 PostgreSQL 的函数机制，pgScript 对内存的使用更合理因而执行效率更高。pgScript 机制之所以能达到这种效果，是因为 PostgreSQL 函数机制会将所有工作在最后一次性批量提交，此前未提交的工作成果都保存在内存中。相比之下，pgScript 在运行脚本时每执行一条 SQL 语句就提交一次，这样就使得 pgScript 特别适合执行会消耗大量内存而又不需要作为一个完整事务来提交的任务。一旦某个事务被提交，则该事务占用的内存会立即被释放，这部分内存即可用于下一个事务。在“使用 pgScript 实现地理编码”这篇博文（http://www.postgresonline.com/journal/archives/181-pgAdmin-pgScript.html）中你可以看到我们所做的示例。

pgScript 脚本语言是一种弱类型语言（即定义变量时无需明确指定其类型），支持条件判断、循环、数据生成器、基本的打印函数以及记录型变量，其语法与微软 SQL Server 数据库所使用的 Transact-SQL 语法类似。前面加 `@` 的是变量，可以存放标量或数组，包括 SQL 命令的执行结果。`DECLARE`、`SET`、`IF-ELSE`、`WHILE` 等语法在 pgScript 中都支持。

可以在 SQL 查询执行窗口中来执行 pgScript。在窗口中输入脚本后，点击 pgScript 图标来执行（）。

以下我们将演示一些 pgScript 脚本的例子。示例 4-1 演示了如何使用 pgScript 记录型变量和循环语法来构建一个交叉表，使用的基础表是 `lu_fact_types`，该表是我们在示例 7-18 中创建的。以下 pgScript 脚本中创建了一个名为 `census.hisp_pop` 的空表，该表有以下数字型列：`hispanic_or_latino`、`white_alone` 和 `black_or_african_american_alone`，以及其他一些列。

示例 4-1：在 pgScript 中使用记录型变量建表

```
DECLARE @I, @labels, @tdef;
SET @I = 0;

变量labels将用于存放记录。
SET @labels =
    SELECT
        quote_ident(
            replace(
                replace(lower(COALESCE(fact_
subcats[4], fact_subcats[3])), ' ', '_'),':',''
            )
        ) As col_name,
    fact_type_id
    FROM census.lu_fact_types
    WHERE category = 'Population' AND fact_subcats[3] ILIKE 'Hispanic or Latino%'
    ORDER BY short_name;

SET @tdef = 'census.hisp_pop(tract_id varchar(11) PRIMARY KEY ';

使用LINES函数来循环遍历每一条记录。
WHILE @I < LINES(@labels)
BEGIN
    SET @tdef = @tdef + ', ' + @labels[@I][0] + ' numeric(12,3) ';
    SET @I = @I + 1;
END

SET  @tdef = @tdef + ')';

打印表def。
PRINT @tdef;
```

```
创建表。
CREATE TABLE @tdef;
```

尽管 pgScript 中没有专门用于执行动态 SQL 的命令，但我们可以通过示例 4-1 中所示的方法来实现执行动态 SQL，即将 SQL 字符串分配给某个变量。我们在下面的示例 4-2 中更加深入地挖掘了 pgScript 的功能，该示例中我们对上面刚刚创建好的 `census.hisp_pop` 表进行了填充操作。

示例 4-2：使用 pgScript 循环填充表

```
DECLARE @I, @labels, @tload, @tcols, @fact_types;
SET @I = 0;
SET @labels =
    SELECT
        quote_ident(
            replace(
                replace(
                    lower(COALESCE(fact_subcats[4], fact_subcats[3])), ' ',
'_'),':',''
            )
        ) As col_name,
    fact_type_id
    FROM census.lu_fact_types
    WHERE category = 'Population' AND fact_subcats[3] ILIKE 'Hispanic or Latino%'
    ORDER BY short_name;

SET @tload = 'tract_id';
SET @tcols = 'tract_id';
SET @fact_types = '-1';

WHILE @I < LINES(@labels)
BEGIN
    SET @tcols = @tcols + ', ' + @labels[@I][0] ;
    SET @tload = @tload +
        ', MAX(CASE WHEN fact_type_id= ' +
        CAST(@labels[@I][1] AS STRING) +
        ' THEN val ELSE NULL END)';
    SET @fact_types = @fact_types + ', ' + CAST(@labels[@I][1] As STRING);
    SET @I = @I + 1;
END

INSERT INTO census.hisp_pop(@tcols)
SELECT @tload FROM census.facts
WHERE fact_type_id IN(@fact_types) AND yr=2010
GROUP BY tract_id;
```

从上面的示例中可以学到的一点是：可以动态地往一个变量中一点点加入 SQL 语句的零碎部分，最终组成一个完整的 SQL 语句。

4.4 以图形化方式解释执行计划

pgAdmin 最为人称道的优秀功能之一就是它能够以图形化方式展示语句执行计划。打开 SQL 语句执行窗口，编写一个 SQL 语句，然后点击“解释查询”图标（）就可以看到此语句的执行计划图示。

例如，我们执行以下查询：

```
SELECT left(tract_id, 5) As county_code, SUM(hispanic_or_latino) As tot,
    SUM(white_alone) As tot_white,
    SUM(COALESCE(hispanic_or_latino,0) - COALESCE(white_alone,0)) AS non_white
FROM census.hisp_pop
GROUP BY county_code
ORDER BY county_code;
```

我们将看到如图 4-16 所示的图形化解释。读懂这种图形化解释有一个小窍门，那就是：尽可能让粗箭头变细！箭头越粗，说明该步骤执行时间越长。

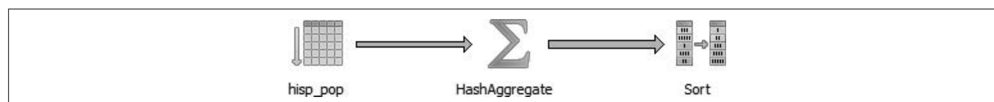


图 4-16：图形化解释示例

如果 SQL 语句执行器菜单栏上的 Query（查询）→Explain（解释）→Buffers（缓冲区）已启用，那么图形化解释就会被禁用。因此切记使用图形化解释时不要启用该选项。除了图形化解释之外，Data Output（数据输出）选项卡上还会显示文本解释计划，本例中的输出如下所示。

```
GroupAggregate  (cost=111.29..151.93 rows=1478 width=20)
       Output:   ("left"((tract_id)::text,    5)),   sum(hispanic_or_latino),
sum(white_alone), ...
  ->  Sort  (cost=111.29..114.98 rows=1478 width=20)
     Output: tract_id, hispanic_or_latino, white_alone,
      ("left"((tract_id)::text, 5)) Sort Key: ("left"((tract_id)::text, 5)) ->
Seq  Scan on census.hisp_pop   (cost=0.00..33.48 rows=1478 width=20)    Output:
tract_id, hispanic_or_latino      ,white_alone, "left"((tract_id)::text, 5)
```

4.5 使用pgAgent执行定时任务

pgAgent 是 PostgreSQL 中执行定时任务的得力工具。同时它也可用于执行操作系统批处理脚本，因此在 Linux/Unix 系统中它可取代 `crontab`；在 Windows 中，它可取代定时任务规划器。事实上，pgAgent 的定时任务功能远比这里描述的更强大：任何一台机器，不管操作系统是什么，只要它上面能安装 pgAgent，那么我们就可以在此机器上执行定时任务。具体步骤是先在这台机器上装好 pgAgent，然后设置该 pgAgent 连接到一个 PostgreSQL 数

据库上，但需要该数据库上预先安装好 pgAgent 所需的功能表和函数。执行定时任务的机器上不需要安装 PostgreSQL 服务端软件，但客户端数据库是必须的，因为要保证 pgAgent 能够连接到外部的 PostgreSQL 服务器。pgAgent 是构建于 PostgreSQL 架构基础上的，因此你可以通过控制服务端的表数据来控制 pgAgent 的行为。例如，如果需要把一个复杂的定时任务复制多次，那么你只需直接登录到 PostgreSQL 服务器并往 pgAgent 的表中插入几条记录即可，完全不需要在 pgAmin 界面上对 pgAgent 执行操作。

本节中我们将教你如何使用 pgAgent。在“设置 pgAgent 来执行定时备份”（http://www.postgresonline.com/journal/archives/19-Setting-up-PgAgent-and-Doing-Scheduled-Backups.html）这篇博文中你可以看到一个真正实用的例子以及设置细节。

4.5.1 安装pgAgent

你可以从 http://www.pgadmin.org/download/pgagent.php 这个链接下载 pgAgent 的安装包。在 Windows 上，你也可以通过 EnterpriseDB 公司提供的 Stackbuilder 软件来安装 pgAgent。安装包中的 SQL 脚本会在 `postgres` 库中自动创建一个名为 pgAgent 的新 schema。然后当你通过 pgAdmin 连到数据库服务器时，可以在目录树上看到一个名为 Jobs（作业）的新节点，如图 4-17 所示。

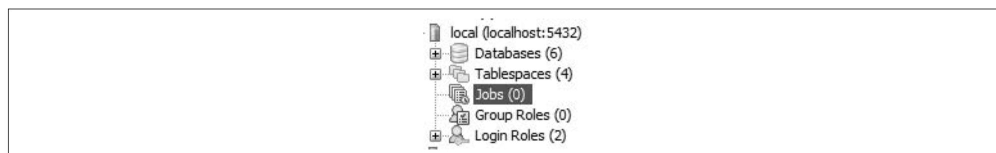


图 4-17：安装了 pgAgent 后的 pgAdmin

如果你希望在其他机器上安装 pgAgent 来执行定时任务，仅需在目标机器上安装 pgAgent 客户端即可，而无需再次执行 pgAgent 自带的 SQL 脚本，因为这个脚本只需在 PostgreSQL 服务端执行一次即可。请特别注意 pgAgent 服务所使用的操作系统账户的权限设置，一定要确保每个 pgAgent 客户端实例都有执行任务所需的权限。



即使批处理任务能以命令行方式执行成功，也并不代表通过 pgAgent 来执行此任务就一定会成功。这一般是因为权限原因导致，pgAgent 总是以 pgAgent 服务所使用的操作系统账户的身份执行规划的任务，如果此账户没有足够权限来执行此批处理任务或者没有访问某些必需路径的权限，那么任务就会失败。

4.5.2 规划定时任务

每个定时任务包含两个组成要素：任务步骤以及执行计划。当创建一个新的任务时，先新

增一个或者多个任务步骤。图 4-18 是新增 / 编辑任务步骤的界面。

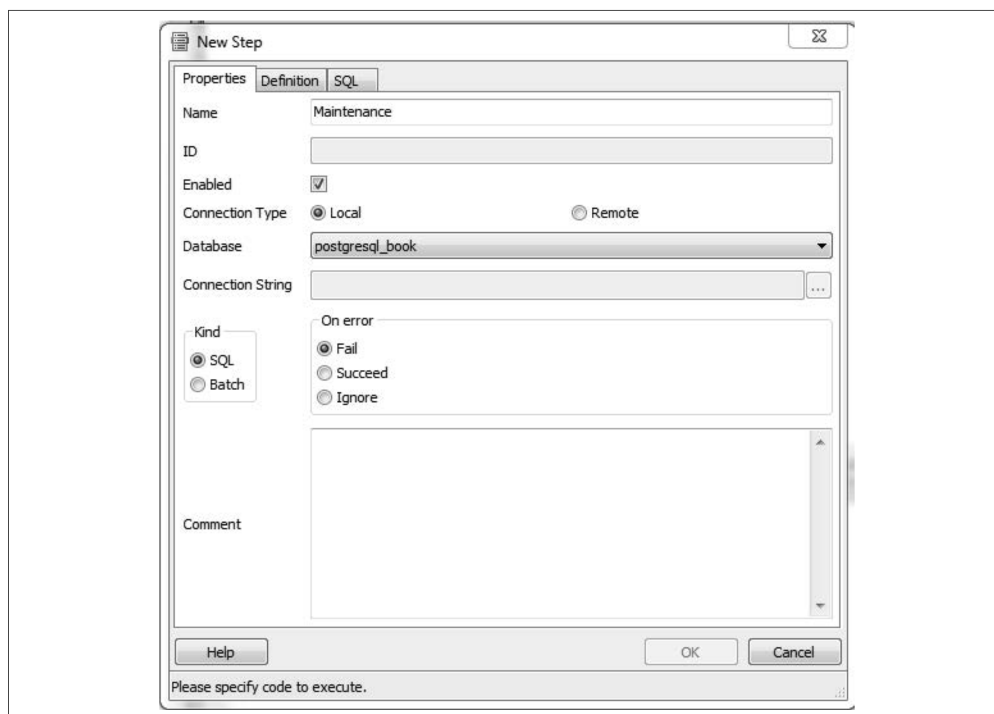


图 4-18：pgAdmin 的分步编辑屏幕

在每一个任务步骤中，你可以设定一条 SQL 语句或者指定一个 shell 脚本作为任务内容，甚至可以复制一整段 shell 脚本作为任务内容。

如果你选择了使用 SQL 语句，那么连接类型选项会变为可用并且默认值会被设为本地，这种情况下该任务步骤中的 SQL 会在 pgAgent 服务端所在的 PostgreSQL 服务器上执行并使用 pgAgent 运行时使用的用户名和密码来进行身份验证。另外还需要指定 pgAgent 在执行此任务时要连到的目标 database。界面上有一个下拉列表供你选择具体的 database。如果你选择了连接到远端数据库服务器，那么供输入连接字符串的文本框会变为可用状态。请在此框中输入完整的连接字符串，包括用于身份验证的信息以及要连到的目标 database。如果你连到一个早期版本的远端 PostgreSQL 数据库，请确保要执行的 SQL 语句的语法在该老版本的 PostgreSQL 上是支持的。

如果你选择了执行批处理任务，那么其语法必须符合执行此任务的操作系统的要求。例如，如果 pgAgent 运行于 Windows 环境下，那么批处理任务脚本必须是合法的 DOS 命令行脚本；如果 pgAgent 运行于 Linux 环境，那么批处理任务脚本必须是合法的 shell 脚本。

多个任务步骤之间是以其名称的字母顺序来排序并执行的。你可以指定每个步骤执行完毕

后的处理方式：如果该步骤执行成功则如何处理；如果该步骤执行失败则如何处理。你可以选择禁用某些步骤而不删除它们，因为你以后还可能重新用上这些步骤。

任务步骤设定好之后，你就可以设定执行计划来执行这些步骤了。通过执行计划页面你可以设定极为复杂的执行策略，你甚至可以设置多个执行计划。

如果你在多台机器上安装了 pgAgent，而这些 pgAgent 都连到同一个 pgAgent 服务端数据库，那么默认情况下所有这些 pgAgent 会执行数据库中记录的所有计划任务。

如果你希望某个计划任务只在某台特定的机器上执行，那么可以在创建计划任务时将页面上的 `host agent`（主机代理）字段设置为希望执行此计划任务的目标主机名。这样其他机器上的 pgAgent 会发现此计划任务的目标主机名与自己所在主机名不符，从而忽略此任务。



pgAgent 包含两部分数据：定义任务的数据以及任务执行日志。这些任务日志会记录在 pgAgent 这个 schema 中，而 pgAgent schema 一般隶属于 `postgres` 数据库。pgAgent 进程会查询待执行的任务信息以决定接下来执行什么任务，然后在执行过程中把相关的任务日志信息写入数据库中。一般来说，用于承载这两类数据的 PostgreSQL 服务器和 pgAgent 是运行于同一台服务器上的，但并不是必须如此，二者可以分离部署。此外，一台 PostgreSQL 服务器可以服务于很多部署在不同主机上的 pgAgent。

一个完整的定时任务看起来如图 4-19 所示。

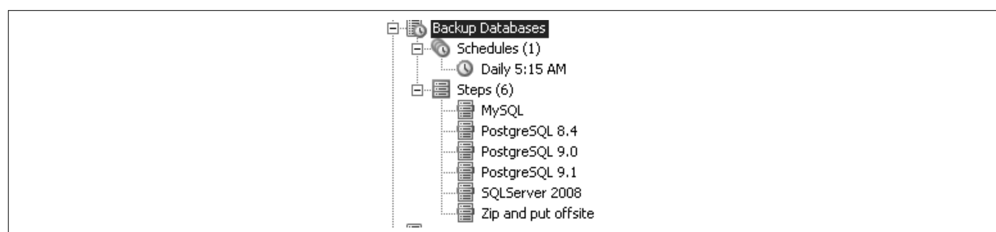


图 4-19：pgAdmin 界面上的 pgAgent 定时任务

4.5.3 一些有用的pgAgent相关查询语句

如果你 SQL 技术高超，那么完全可以通过直接修改 pgAgent 的元数据表来实现对定时任务的复制、删除和修改。只是在修改时要特别小心谨慎，不要搞错！例如，要想查看控制着 pgAgent 和定时任务具体行为的后台表内容，只需连到 `postgres` 数据库并执行以下示例 4-3 中的查询。

示例 4-3：查询 pgAgent 相关表的描述信息

```
SELECT c.relname As table_name, d.description
FROM
    pg_class As c INNER JOIN
    pg_namespace n ON n.oid = c.relnamespace INNER JOIN
    pg_description As d ON d.objoid = c.oid AND d.objsubid = 0
WHERE n.nspname = 'pgagent'
ORDER BY c.relname;

table_name      | description
---------------+-------------------------
pga_job         | Job main entry
pga_jobagent    | Active job agents
pga_jobclass    | Job classification
pga_joblog      | Job run logs.
pga_jobstep     | Job step to be executed
pga_jobsteplog  | Job step run logs.
pga_schedule    | Job schedule exceptions
```

尽管在 pgAdmin 中制定 pgAgent 定时任务和观察其执行日志的界面已经非常直观易懂，但如果你设置了很多定时任务或者你希望查看自己的定时任务执行结果总览，那么就会非常需要生成自己的定时任务执行报告。示例 4-4 演示了我们在此情况下经常会使用的一个查询语句。

示例 4-4：列出从今天开始的定时任务执行结果

```
SELECT j.jobname, s.jstname, l.jslstart,l.jslduration, l.jsloutput
FROM
    pgagent.pga_jobsteplog As l INNER JOIN
    pgagent.pga_jobstep As s ON s.jstid = l.jsljstid INNER JOIN
    pgagent.pga_job As j ON j.jobid = s.jstjobid
WHERE jslstart > CURRENT_DATE
ORDER BY j.jobname, s.jstname, l.jslstart DESC;
```

有时候定时任务即使失败了也会报成功，因为 pgAgent 并不总能准确判断 shell 脚本的执行结果到底是成功还是失败。日志中的 `jsloutput` 字段提供 shell 输出，该输出通常会详细说明哪里出现了错误。



在 Windows 平台上，有若干个版本的 pgAgent 经常会把事实上已执行成功的 shell 脚本的执行结果判定为执行失败。如果你遇到了这种情况，那么应该把任务步骤的结果处理方式设定为“错误时忽略”。这是一个已知的 bug，我们希望在将来的版本中能够尽快修复。

第5章 数据类型

与其他所有数据库一样，PostgreSQL 也支持数字型、字符串型、日期和时间型以及布尔型等业界常用的数据类型。 但 PostgreSQL 的先进之处在于它还支持数组、带时区的日期时间、时间间隔、区间、JSON、XML 以及其他很多数据类型，此外还支持用户自定义数据类型。本章们不会一一介绍 PostgreSQL 支持的所有数据类型，如果你需要了解的话，请自行参考官方手册。我们将着重介绍 PostgreSQL 独有的若干数据类型以及在那些通用数据类型方面 PostgreSQL 与业界其他数据库有哪些细微差异。

如果离开了相应的函数和运算符，数据类型将完全无用武之地。PostgreSQL 为各种数据类型提供了功能强大种类丰富的原生函数和运算符支持，而且很多扩展包还在源源不断地提供新的扩展。我们将在本章中介绍一些使用比较广泛的类型。



我们所说的函数是指 `f(x)` 这种形式的函数；我们所说的“运算符”是指一些字符性的运算符号，比如 `+`、`-`、`*`、`/` 这类，具体可分为需要单个实参的一元运算符和需要两个实参的二元运算符。最简单的运算符是需要一个或多个实参的函数的一个符号别名，比如 `+` 这个运算符可以理解为 `add(x,y)` 这个函数的一个符号别名。当使用运算符时，请记住它在面对不同的数据类型时所代表的含义是不同的。比如加号对数字来说就是相加，对区间类型来说就是区间的并集。

5.1 数值类型

PostgreSQL 支持常用的整数、小数、浮点数等数字类型。我们想重点介绍的是 serial 类型

以及一个灵活实用的整数序列生成函数。

5.1.1　serial类型

serial 类型和它的兄弟类型 bigserial 是两种可以自动生成递增整数值的数据类型，一般如果表本身的字段不适宜作为主键字段时，会增加一个专门的字段并指定为 serial 类型以作为主键。在不同的数据库产品中这种数据类型有着不同的称呼，一般最常见的叫法是 autonumber。建表时如果指定了一个字段类型为 serial，那么 PostgreSQL 会首先将其作为整型处理，同时自动在该表所在 schema 中创建一个名为 table_name_column_name_seq 的序列。然后设定该序列为该整型字段的取值来源。如果修改了表定义并删除此 serial 字段，那么系统同时也会自动删除掉附属的序列。

在 PostgreSQL 中，序列自身就是一种数据库资产。你可以通过 pgAdmin 图形界面或者 ALTER SEQUENCE 语句来管理该类对象。你可以设置其当前值和边界值（也就是最大和最小值），还可以设置每次递增的步长。虽然一般来说序列值都是递增的，但你也可以将其设为递减，只要将步长值 increment 设为负数即可。作为一种独立的数据库资产，你可以通过 CREATE SEQUENCE 命令来创建序列，还可以在多张表间共用同一个序列。如果你需要生成一个跨越多表的唯一键值，那么这种多表共享序列的用法是特别方便的。

要实现多表共用同一个序列，请先将字段定义为 integer 或者 bigint 类型，然后指定其默认值为 nextval(sequence_name) 即可。



如果你重命名了一张含 serial 字段的表，这张表的 serial 字段关联的序列是不会跟着改名的，但关联运作机制是不受影响的。如果你特别重视对象名的一致性，可以手动修改序列的名称以保持与表名一致。

5.1.2　生成数组序列的函数

PostgreSQL 有一个名为 generate_series 的灵活又实用的数组生成函数，目前为止我们还没发现有哪种数据库支持类似功能。generate_series 函数的方便之处在于你可以使用它有效地模仿 SQL 中的 for 循环。假设我们需要得到一个列表，该列表中包含了特定日期区间内每个月份最后一天的日期。如果不借助 generate_series 函数，要实现此功能要么写一个内部使用循环的函数，要么生成一个基于日期类型的笛卡儿积然后逐条筛选。如果使用了 generate_series 函数，那么仅需一行 SQL 即可。我们会在后续的示例 5-11 中展示具体做法。

示例 5-1 使用可选的步长形参来生成整数序列。

示例 5-1：使用 generate_series() 函数生成步长为 13 的整数序列

```
SELECT x FROM generate_series(1,51,13) As x;
```

```
x
----
1
14
27
40
```

如上面的示例 5-1 所示，你可以传入一个可选的步长实参来指定对于每个后续元素要跳过多少个步长。如果不指定，则默认为 1。另外请注意：结束值将永远不会超出我们指定的区间，因此，尽管我们的区间结束于 51，但我们的最后一个数字是 40，因为 40 再加上 13 就会超出上限。

5.2 字符和字符串

PostgreSQL 有三种最基础的数据类型：character（也称为 char）、character varying（也称为 varchar）和 text。varchar 和 text 适用于存储长度可变化的文本，每一行记录需要多大空间就分配多大空间。这两种类型的存储方式是完全一致的，性能表现也没有差别。

char 类型占用的存储空间是固定的，适用于如邮政编码、电话号码以及社会保险号等定长字符串的存储。如果存储的字符长度达不到 char 类型的定义长度，那么会在后面用空格填充，不管存储时还是查询显示时都是这样。这种模式对于存储空间有所浪费，但这也是 ANSI SQL 标准中规定的做法。除此以外，在 PostgreSQL 中 char 和 varchar 没有别的性能差别。

没有大小修饰符的 varchar 与 text 之间几乎没什么差别。对于 text 列来说，不管它所包含的字符有多少，你都可以对其进行排序。有些数据库驱动程序（比如 ODBC）可能会对二者的处理略有差别。varchar 和 text 的存储空间上限均为约 1 GB，但事实上系统在后台会把超过一个物理存储页大小的内容用 TOAST 机制处理。



在 PostgreSQL 9.2 版之前，如果你需要加大一个 varchar 字段的长度定义而此时该表中已有记录存在，那么系统实际上会在后台建一张新表再把数据从旧表转移到新表，该过程会耗费一定时间并会导致锁表。因此，大家一般会通过使用 text 类型来避免此问题。

关于是否应该彻底废弃 varchar 并完全转用 text 这个问题，业界一直有着不同的声音。我们不会在此处浪费篇幅争论此问题，请你参考“我为 varchar(x) 的辩护”这篇博文（http://www.postgresonline.com/journal/archives/154-In-Defense-of-varcharx.html）以了解这场争论的详情。

有时候，为了保持跨平台应用的兼容性，你需要使字符串类型的操作变得不区分大小写。要实现此目标，你需要重写那些区分大小写的比较运算符。相比 text，对 varchar 进行

运算符重写要更容易一些。我们在“将 MS Access 与 PostgreSQL 协同使用”这篇博文中（http://www.postgresonline.com/journal/archives/24-Using-MS-Access-with-PostgreSQL.html）演示了如何使 `varchar` 类型变得不区分大小写而且同时还能够用上索引。

5.2.1 字符串函数

常见的字符串操作包括：填充（`lpad`、`rpad`）、修整空白（`rtrim`、`ltrim`、`trim`、`btrim`）、提取子字符串（`substring`）以及连接（`||`）。示例 5-2 演示了填充操作，示例 5-3 演示了修整空白操作。

示例 5-2：使用 `lpad` 和 `rpad` 进行填充操作

```
SELECT lpad('ab', 4, '0') As ab_lpad, rpad('ab', 4, '0') As ab_rpad, lpad('abcde',
4, '0') As ab_lpad_trunc; ❶

ab_lpad | ab_rpad | ab_lpad_trunc
--------+---------+-----
00ab    | ab00    | abcd
```

❶ 如果字符串超过指定长度的话，`lpad` 不但不会填充，反而会对其进行截断。

默认情况下，`trim` 函数用于移除空格，但你也可以传入一个可选实参，指示要剪裁的其他字符。

示例 5-3：剪裁空格和字符

```
SELECT
a As a_before, trim(a) As a_trim, rtrim(a) As a_rt,
i As i_before, ltrim(i, '0') As i_lt_0,
rtrim(i, '0') As i_rt_0, trim(i, '0') As i_t_0
FROM ( SELECT repeat(' ', 4) || i || repeat(' ', 4) As a, '0' || i As i FROM gener
ate_series(0, 200, 50) As i
) As x;

a_before     | a_trim |  a_rt    | i_before | i_lt_0 | i_rt_0 | i_t_0
-------------+--------+---------+----------+--------+--------+-------
     0       | 0      |     0   | 00       |        |        |
     50      | 50     |     50  | 050      | 50     | 05     | 5
     100     | 100    |     100 | 0100     | 100    | 01     | 1
     150     | 150    |     150 | 0150     | 150    | 015    | 15
     200     | 200    |     200 | 0200     | 200    | 02     | 2
```

从 PostgreSQL 9.0 版开始支持一种很有用的字符串操作函数 `string_agg`，我们已在示例 3-11 中展示了其用法，你还将在示例 5-21 中再次见到它。该函数与 MySQL 的 `group_concat` 函数作用是相同的。

5.2.2 将字符串拆分为数组、表或者子字符串

PostgreSQL 中有一些函数可以对字符串进行拆分操作。

split_part 函数可以将指定位置的元素从用固定分隔符分隔的字符串中取出来，如示例 5-4 所示。

示例 5-4：取出分隔符字符串中的第 *n* 个元素

```
SELECT split_part('abc.123.z45', '.', 2) As x;

x
--------
123
```

string_to_array 函数可以将基于固定分隔符的字符串拆分为一个数组。通过将 string_to_array 和 unnest 函数结合使用，你可以将一个字符串展开为若干记录行，如示例 5-5 所示。

示例 5-5：将基于固定分隔符格式的字符串展开为记录行

```
SELECT unnest(string_to_array('abc.123.z45', '.')) As x;

x
--------
abc
123
z45
```

5.2.3　正则表达式和模式匹配

PostgreSQL 对正则表达式的支持是极其强大的。你可以设定查询返回结果的格式为表或者数组，并且对其进行极其复杂的替换和更新操作。包括逆向引用（back reference）在内的一些高级搜索方法都是支持的。在本节中，我们将提供一个小型的示例以说明这些内容。如需了解更多相关信息，请参考 PostgreSQL 官方手册中的“模式匹配”（http://www.postgresql.org/docs/current/interactive/functions-matching.html）和“字符串函数”（http://www.postgresql.org/docs/current/interactive/functions-string.html）这两节的内容。

示例 5-6 展示了如何对以数字形式存储的电话号码进行格式化操作。

示例 5-6：使用逆向引用技术对电话号码进行重新格式化

```
SELECT regexp_replace(
'6197306254',
'([0-9]{3})([0-9]{3})([0-9]{4})',
 E'\(\\1\) \\2-\\3'
 ) As x;

x
--------------
(619) 730-6254
```

\\1 和 \\2 是模式匹配表达式中的元素。我们使用反斜杠（\）来转义圆括号，表示此处是

真的要作为一个圆括号来用。E' 是 PostgreSQL 的构造符语法，表示后续跟着的字符串是一个表达式，其中类似 \ 的特殊字符应该按照字面含义来处理。

假设一个字符串中内嵌了一些电话号码，示例 5-7 演示了如何仅通过一个 SQL 语句就将这些号码提取出来并作为记录行输出：

示例 5-7：将文本中的电话号码作为单独的行返回

```
SELECT unnest(regexp_matches( 'Cell (619)852-5083. Casa 619-730-6254. Bésame mucho.
        E'[(]{0,1}[0-9]{3}[)-.]{0,1}[0-9]{3}[-.]{0,1}[0-9]{4}', 'g')
) As x;

x
-------------
(619)852-5083
619-730-6254
```

示例 5-7 中用到的匹配规则如下所示。

- `[(]{0,1}`：开始是 0 个或者 1 个 (。
- `[0-9]{3}`：跟着 3 位数字。
- `[)-.]{0,1}`：跟着 0 个或者 1 个) 或者 - 或者 .。
- `[0-9]{4}`：跟着 4 位数字。
- `regexp_matches` 函数会返回根据一个正则表达式筛选匹配得到的字符串数组。如果不传入 `g` 形参，则仅返回第一个命中的字符串。`g` 表示 global，即需要进行完整搜索并返回所有匹配上的字符串，每个字符串作为数组中的一个元素。
- `unnest` 函数将一个数组分解为一个行集。



同一个正则表达式可以有多种写法。比如，`\\d` 代表 `[0-9]`。但我们建议不要为了省那几个字符而把表达式搞得太晦涩难懂，应该采用更加容易理解的写法。

除了正则表达式专用的那些函数外，你还可以将正则表达式与 `SIMILAR TO`（`~`）运算符一起使用。以下查询可以查出所有内嵌了电话号码的字符串：

```
SELECT description
FROM mytable
WHERE description ~ E'[(]{0,1}[0-9]{3}[)-.]{0,1}[0-9]{3}[-.]{0,1}[0-9]{4}';
```

5.3 时间类型

PostgreSQL 对时间类型的支持在业界是无人能及的。除了常见的日期和时间类型，PostgreSQL 还支持时区，并能够按照不同的时区对夏令时进行自动转换。此外 PostgreSQL

还支持一些特殊的数据类型，如 `interval`，该类型可以用于对日期时间进行数学运算。PostgreSQL 还有正无穷大和负无穷大的概念，这样我们就不用为了表达这两个概念而弄出一些奇奇怪怪的潜规则，搞这些潜规则迟早会导致问题。最后，9.2 版引入了对区间（range）类型的支持，该类型可以表达时间区间的概念，并且提供了大量与区间运算相关的运算符、函数和索引。我们会在本章后续的 5.5 节中详细介绍该类型。

在最新的版本中，PostgreSQL 支持 9 种时间相关的数据类型。理解这些类型之间的区别很重要，否则你就无法为不同业务场景选择适用的数据类型。除了 `range` 类型外，其他所有类型都遵循 ANSI SQL 标准。业界的其他数据库最多支持这些类型中的一部分而非所有。Oracle 支持的类型最多，SQL Server 其次，MySQL 又次之，MySQL 的任何版本都不支持时区类型。

PostgreSQL 的每种时间类型都有其独特之处，因此我们接下来对其分别做一些更详细的介绍。

- `date`

  该类型仅存储月、日、年，没有时区、小时、分和秒的信息。

- `time`（又称`time without time zone`）

  该类型仅存储小时、分、秒信息，不带日期和时区信息。

- `timestamp`（又称`timestamp without time zone`）

  该类型存储了日期（年、月、日）和时间（时、分、秒）数据，但不带时区信息。因此，即使你修改了数据库服务器所在的时区信息，该类字段查询出来显示的值也是固定不变的。

- `timestamptz`（又称`timestamp with time zone`）

  该类型同时存储了日期、时间以及时区信息。在系统内部，该类型的字段值是以 UTC 世界标准时间格式存储的，但当查询显示时，会按照服务器的时区设置进行换算后再显示（时区也可以在库级 / 用户级 / 会话级分别进行设置）。如果你输入的时间戳不带时区数据，那么存入 `timestamptz` 类型字段中时，PostgreSQL 会自动使用当前数据库服务器的时区信息来补充。如果修改了数据库服务器的时区设置，你可以看到查询出来的时间数据发生了变化。

- `timetz`（又称`time with time zone`）

  与 `timestamptz` 类型类似，但该类型的使用频率较低，因为它虽然携带了时区信息但却没有日期信息。该类型永远假设当前时间是夏令时。有的编程语言不支持这种仅有时间而无日期的数据类型，因此可能会将其自动转换为带时区的时间戳类型，转换时日期就取计算机系统时间的初始值（例如，Unix 时间纪元起始于 1970 年，因此转换后的日期就是 1970 年 1 月 1 日，时区和时间不变，夏令时）。

- `interval`

  该类型描述了一个时间段的长度，单位可以是小时、天、月、分钟或者其他粒度。该类型适用于对日期和时间进行数学运算的场景。例如，假设从现在开始 666 天之后世界就会灭亡，那么你可以在现在的时刻上加上长度为 666 天的一个 `interval` 类型值就可以知道世界灭亡的准确时刻。

- `tsrange`

  该类型是 9.2 版新引入的，可用于定义 `timestamp with no timezone` 的开区间和闭区间。该类型包含两个时间戳以及开区间和闭区间限定符。例如，`'[2012-01-01 14:00 2012-01-01 15:00)'::tsrange` 定义了从 14:00 开始到 15:00 之前结束的一个时间段。请参考 PostgreSQL 官方手册中“区间类型”这一节（http://www.postgresql.org/docs/current/static/rangetypes.html#RANGETYPES-BUILTIN）以了解更多信息。

- `tstzrange`

  该类型也是 9.2 版新引入的，可用于定义 `timestamp with timezone` 的开区间和闭区间。

- `daterange`

  该类型也是 9.2 版引入的，可用于定义日期的开区间和闭区间。

5.3.1 时区详解

PostgreSQL 中有众多支持时区的数据类型，关于它们有一个常见的误解，就是认为 PostgreSQL 会在日期和时间类型的基础上额外增加一个标记来标识时区，这种理解是错误的。如果你存储了这么一个带时区的信息：`2012-2-14 18:08:00-8`（–8 代表比 UTC 时间迟 8 小时的时区），PostgreSQL 内部其实是这么工作的。

(1) 通过计算得到 2012-02-14 18:08:00-8 代表的 UTC 标准时间，就是 2012-02-15 04:08:00-0。
(2) 把上述计算得到的 UTC 标准时间存储下来。

当你回调该数据以用于显示时，PostgreSQL 内部是这样运作的：

(1) 找到服务器所观察到的时区或者请求的时区（例如 `America/New_York`）。
(2) 计算该时区相对于 UTC 标准时间的时差（对于 `America/New_York` 时区来说，与 UTC 的时差是 –5 小时）。
(3) 根据 UTC 标准时间和时区的时差计算出当地时间（2012-02-15 16:08:00 加上时差 –5 小时后得到 2012-02- 15 21:08:00）。
(4) 显示计算结果（`2012-02- 15 21:08:00`）。

可以看到，PostgreSQL 并没有存储时区信息而仅是使用时区信息来把日期和时间转换为

UTC 标准时间再存储下来。此后时区信息就丢失了。当 PostgreSQL 需要显示该日期时间信息时，它会按顺序查找当前会话级、用户级、数据库级、服务器级的时区设置，然后使用找到的第一个时区来将 UTC 标准时间转换为对应时区的时间值后再显示。如果你使用了带时区信息的数据类型，请务必要了解将服务器从一个时区搬迁到另一个时区后将会发生的后果。假设你的数据库服务器起初在纽约，然后将其数据拿到洛杉矶做了恢复，那么所有带时区信息的日期和时间数据看起来都会不一样了。这初看起来有点怪，但其实是正常的，你务必要预见到这种情况的发生。

以下我们将向你演示一个时区处理不当导致出问题的例子。假设麦当劳公司的服务器都部署在东海岸，服务器中记录了各门店的开门营业时间，并且是用 `timetz` 格式存储的。然后在旧金山开了一家新的麦当劳分店，分店的经理给麦当劳总部打电话，告知他们要求把新店的信息纳入总部的管理数据库，并标记其开门营业时间为上午 7 点。于是位于东海岸的数据库服务器中会记录下分店的营业时间为上午 7 点，但这个时间转换到旧金山当地时区却是凌晨 4 点。于是旧金山很多早起的人们就会很奇怪为什么明明说好了是凌晨 4 点开门却到了时间还没营业。买不到早点是小事，但你可以想象这三小时的时差能导致多么大的混乱，这甚至可能导致人命关天的问题。

看了上面的例子，你可能会问：既然这么危险，那么为什么还要使用带时区的时间类型？原因有以下几个：首先，这些类型能够自动执行时区转换，从而避免了繁琐的手工劳动。例如，某航空公司的某个航班上午 8 点从波士顿出发，上午 11 点到达洛杉矶，但是该公司的数据库服务器位于欧洲，如果这些时间入库时都要手工计算时差后再录入那就效率太低了。使用了支持时区的数据类型后只需要录入带时区信息的波士顿和洛杉矶本地时间即可；另一个使用带时区的数据类型的理由是其自动处理夏令时的能力。世界各国对于夏令时的规定五花八门，如果某个数据库可能会被全球各地的应用访问，那么就需要及时按照最新的全球夏令时规定来更新库中的时间信息。手工跟踪全球夏令时的变化是一件无比繁琐的工作，这需要一个全职的程序员专门来收集各国的夏令时安排，并在前述数据库中刷新这些国家（包括其海外飞地）的相关时间数据。

这里有一个非常有趣的例子：一位出差中的销售员需要坐飞机回家，起点是旧金山，终点是奥克兰附近。当她登上飞机时，当地时钟显示的时间是 2012 年 3 月 11 日凌晨 1 点 50 分。当她降落时，当地时钟显示时间是 2012 年 3 月 11 日凌晨 3 点 10 分。那么请问这段旅程共花了多长时间？要回到这个问题有一个关键点，那就是在这段飞行的过程中发生了夏令时的转换，也就是说时间向前跃迁了。如果使用了带时区信息的时间戳，算出来的时间间隔就是 20 分钟，对于一段仅仅跨越旧金山海湾的短途飞行来说，这个答案显然是可信的。如果我们不使用带时区信息的数据类型，一定会得到错误的答案。

```
SELECT '2012-03-11 3:10 AM America/Los_Angeles'::timestamptz
 - '2012-03-11 1:50 AM America/Los_Angeles'::timestamptz;
```

以上查询得到的答案是 20 分钟，然而以下查询得到的答案却是 1 小时 20 分钟。

```
SELECT '2012-03-11 3:10 AM'::timestamp- '2012-03-11 1:50 AM'::timestamp;
```

我们再举几个例子来把这个问题讲得更透彻一些。如示例 5-8 所示，我输入时使用的是带时区的洛杉矶本地时间，但由于数据库服务器位于波士顿，所以查询时输出的时间是带时区信息的波士顿本地时间。请注意，输出显示附带了时差，这是没问题的，与我原始录入的时间之间仅仅是显示差异而已，在数据库系统内部是以 UTC 标准时间存储的。

示例 5-8：输入时使用的是一个时区的本地时间，输出却是另一个时区的本地时间

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz;

2012-02-29 01:00:00-05
```

在示例 5-9 中，我们要求返回的是不带时区的时间戳。因此这个查询在全世界任何地方的数据库服务器上执行都会返回相同的结果。

示例 5-9：将带时区信息的时间戳数据转换为不带时区的时间戳数据

```
SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz AT TIME ZONE 'Europe/
Paris';

2012-02-29 07:00:00
```

以上查询其实回答了这么一个问题：洛杉矶本地时间 2012-02-28 10:00 p.m. 对巴黎来说是当地时间几点？请注意查询结果是不带相对于 UTC 标准时间的时差的。另外请注意可以通过官方名称而非 UTC 时差来指定一个时区，可以访问维基百科来查看所有时区的官方名称（http://en.wikipedia.org/wiki/Tz_database）。

5.3.2 日期时间类型的运算符和函数

时间间隔类型（`interval`）的引入极大简化了 PostgreSQL 中日期和时间类型的数学运算过程。如果没有 `interval` 类型，我们就得创建一堆专门的函数来实现这些运算功能，很多其他数据库就是这么干的。通过 `interval` 类型，我们可以使用我们很熟悉的加减运算符对日期和时间进行进行相加或者相减操作。下面的例子展示了可用于日期和时间类型的运算符和函数。

+ 运算符可以在一个时间类型值上加上一段时间间隔：

```
SELECT '2012-02-10 11:00 PM'::timestamp + interval '1 hour';

2012-02-11 00:00:00
```

你也可以将两个 `interval` 类型直接相加：

```
SELECT '23 hours 20 minutes'::interval + '1 hour'::interval;

24:20:00
```

- 运算符可以从一个时间类型值中减去一段时间间隔：

```
SELECT '2012-02-10 11:00 PM'::timestamptz - interval '1 hour';

2012-02-10 22:00:00-05
```

示例 5-10 中展示了区间重叠运算符 OVERLAPS 的用法，如果两个参与运算的时间段有重叠，那么判定结果就是 true。这是 ANSI SQL 标准中规定的运算符，其效果等价于 overlaps 函数。OVERLAPS 运算符需要四个形参，前两个是第一个时间段的首尾时间点，后两个是第二个时间段的首尾时间点。OVERLAPS 运算符会将这两个时间段看作是半开半闭区间，也就是说起始时点包含在时段内，结束时点不包含在时段内。这与 BETWEEN 运算符的逻辑是不一样的，BETWEEN 会认为起始点和结束点都是包含在区间内的。只要你设置的时间段的起始时点和结束时点不相同（如果相同的话就意味着时间段长度为 0，也就说时间段变成了一个时间点），这个差异不会造成什么问题。如果你经常需要使用 OVERLAPS 运算符，请务必注意这一点。

示例 5-10：对时间戳和日期类型使用 OVERLAPS 运算符

```
SELECT ('2012-10-25 10:00 AM'::timestamp, '2012-10-25 2:00 PM'::timestamp) OVERLAPS
('2012-10-25 11:00 AM'::timestamp,'2012-10-26 2:00 PM'::timestamp) AS x,
('2012-10-25'::date,'2012-10-26'::date) OVERLAPS
('2012-10-26'::date,'2012-10-27'::date) As y;

x  |y
---+---
t  |f
```

除了运算符以外，PostgreSQL 还支持一些时间类型的函数。你可以从 PostgreSQL 官方手册的“日期和时间类型的相关函数和操作”（http://www.postgresql.org/docs/current/interactive/functions-datetime.html）这一节内容中查到完整的函数列表，我们在此仅演示一个例子。

我们再次用到了用途广泛的 generate_series 函数。你可以对日期时间类型使用此函数，此时应使用 interval 类型值作为步长。

如示例 5-11 所示，我们可以用本地日期时间格式来输入日期，也可以使用在国际上更为通用的 ISO 格式“Y-M-D”来输入日期。PostgreSQL 会自动识别不同的输入格式。为保险起见，我们倾向于使用 ISO 标准格式，因为在不同文化中对于日期的惯用格式是不一样的。由于本地设置的差异，在数据库服务器之间甚至是数据库实例之间也会存在这种日期格式差异。

实例 5-11：使用 generate_series() 函数来生成时间序列数组

```
SELECT (dt - interval '1 day')::date As eom
FROM generate_series('2/1/2012', '6/30/2012', interval '1 month') As dt;
```

```
eom
------------
2012-01-31
2012-02-29
2012-03-31
2012-04-30
2012-05-31
```

另一种经常使用的操作就是从日期和时间类型的数值中抽取出一部分。在 PostgreSQL 中，联用 `date_part` 和 `to_char` 函数可以实现此目标。示例 5-12 中，除了演示这两个函数的用法外，我们还顺便为你演示了一下带时区信息的日期时间类型在发生夏令时变换时的转换逻辑，为此我们特地挑选了一个美国东部时区（`US/East`）中横跨夏令时变化点的时间段。夏令时从凌晨 2 点生效，因此该表的最后一行就是夏令时变化以后的新时间。

示例 5-12：从日期时间类型中提取部分元素

```
SELECT dt, date_part('hour',dt) As mh, to_char(dt, 'HH12:MI AM') As tm
FROM
generate_series( '2012-03-11 12:30 AM', '2012-03-11 3:00 AM', interval '15 minutes'
) As dt;

dt                     | mh | tm
-----------------------+----+----------
2012-03-11 00:30:00-05 |  0 | 12:30 AM
2012-03-11 00:45:00-05 |  0 | 12:45 AM
2012-03-11 01:00:00-05 |  1 | 01:00 AM
2012-03-11 01:15:00-05 |  1 | 01:15 AM
2012-03-11 01:30:00-05 |  1 | 01:30 AM
2012-03-11 01:45:00-05 |  1 | 01:45 AM
2012-03-11 03:00:00-04 |  3 | 03:00 AM
```

`generate_series` 函数默认生成的是 `timesatamptz` 类型数据，需要显式转换为 `timestamp` 类型。

5.4 数组类型

数组在 PostgreSQL 中扮演着重要的角色。它在构造聚合函数、形成 `IN` 和 `ANY` 子句、承载数据类型转换过程中生成的中间值等领域发挥着重要作用。在 PostgreSQL 中，每种数据类型都有相应的以该类型为基础的数组类型。如果你自定义了一个数据类型，那么 PostgreSQL 会在后台自动为此类型创建一个数组类型。例如，`integer` 整数类型有一个相应的整数数组类型 `integer[]`，字符类型 `character` 也有相应的字符数组类型 `character[]`，以此类推。以下我们将向你展示一些可以快速构造出数组的函数，可以免除你一个个元素录入的麻烦。此外还有一些用于管理数组的函数。你可以从 PostgreSQL 官方手册的“数组函数和运算符”一节（http://www.postgresql.org/docs/current/interactive/functions-array.html）中查到全部的数组函数和运算符列表。

5.4.1 数组构造函数

最基本的构造数组的方法就是一个个元素手动录入，语法如下：

```
SELECT ARRAY[2001, 2002, 2003] As yrs;
```

如果数组元素存在于一个查询返回的结果集中，那么可以使用这个略复杂一些的构造函数 `array()` 来生成数组：

```
SELECT array(
SELECT DISTINCT date_part('year', log_ts) FROM logs ORDER BY date_part('year',
log_ts)
);
```

尽管 `array` 函数仅能用于将单字段的查询结果集转换为数组，但你依然可以指定一个复合数据类型作为查询结果，这种情况下可以获得多列结果。我们会在本章后续的 5.8 节中演示该用法。

你可以把一个直接以字符串格式书写的数组转换为一个真正的数组，语法如下：

```
SELECT '{Alex,Sonia}'::text[] As name, '{43,40}'::smallint[] As age;

name         | age
-------------+--------
{Alex,Sonia} | {43,40}
```

你还可以用 `string_to_array` 函数将一个用固定分隔符分隔的字符串转换为数组，如示例 5-13 所示。

示例 5-13：将一个分隔符格式的字符串转换为数组

```
SELECT string_to_array('ca.ma.tx', '.') As estados;

estados
----------
{ca,ma,tx}
```

`array_agg` 是一种变型聚合函数，它可以采用一组任何类型的数据并将其转换为数组，如示例 5-14 所示。

示例 5-14：`array_agg` 函数的使用

```
SELECT array_agg(log_ts ORDER BY log_ts) As x
FROM logs
WHERE log_ts BETWEEN '2011-01-01'::timestamptz AND '2011-01-15'::timestamptz;

x
------------------------------------------
{'2011-01-01', '2011-01-13', '2011-01-14'}
```

5.4.2 引用数组中的元素

一般来说，我们会通过数组下标来引用数组元素，请特别注意 PostgreSQL 的数组下标从 1 开始。如果你试图越界访问一个数组，也就是说数组下标已经超过了数组元素的个数，那么不会返回错误，而是会得到一个空值 NULL。下面的例子演示了获取数组的第一个和最后一个元素的方法：

```
SELECT fact_subcats[1] AS primero,
 fact_subcats[array_upper(fact_subcats, 1)] As segundo
FROM census.lu_fact_types;
```

我们使用 array_upper 函数来获取数组元素的个数，该函数的第二个必需的形参代表数组的维度。在本例中，数组是一维的，但 PostgreSQL 支持多维数组。

5.4.3 数组的拆分与连接

PostgreSQL 支持使用 start:end 语法对数组进行拆分。操作结果是原数组的一个子数组。例如，如果要得到一个仅包含当前数组第 2 个至第 4 个元素的新数组，可以使用以下语法：

```
SELECT fact_subcats[2:4] FROM census.lu_fact_types;
```

如果要将两个数组连接到一起，可以使用连接运算符 ||：

```
SELECT fact_subcats[1:2] || fact_subcats[3:4] FROM census.lu_fact_types;
```

5.4.4 将数组元素展开为记录行

另外一个常用的数组操作函数是 unnest，通过它可以将数组元素纵向展开成一个包含若干条记录的结果集，如示例 5-15 所示。

示例 5-15：使用 unnest 函数将数组纵向展开

```
SELECT unnest('{XOX,OXO,XOX}'::char(3)[]) As tic_tac_toe;

tic_tac_toe
---
XOX
OXO
XOX
```

你可以在一个 SELECT 语句中使用多个 unnest 函数，但如果每个 unnest 展开后的记录行数不一致，或者说“对不齐”，那么得到的最终结果将是这些结果集之间的笛卡儿积，看起来不太好理解。

示例 5-16 演示了一个 unnest 展开后可对齐的结果集，也就是说每个 unnest 都输出 3 行记

录，最终连接成的记录也是 3 行，这也是我们一般希望见到的结果。

示例 5-16：多个可对齐数组的展开效果

```
SELECT
unnest('{three,blind,mice}'::text[]) As  t,
unnest('{1,2,3}'::smallint[]) As i;

t     |i
------+-
three |1
blind |2
mice  |3
```

如果你从上述一个数组中拿掉一个元素，那么两个数组的元素就无法对齐了，此时展开得到的结果如示例 5-17 所示。

示例 5-17：多个无法对齐的数组展开后的效果

```
SELECT
unnest( '{blind,mouse}'::varchar[]) As v,
unnest('{1,2,3}'::smallint[]) As i;

v     |i
------+-
blind |1
mouse |2
blind |3
mouse |1
blind |2
mouse |3
```

PostgreSQL 9.4 版引入了一个多实参 `unnest` 函数，该函数会在数组不平衡的位置置入空占位符。新的 `unnest` 的主要缺点是，它仅可以在 `FROM` 子句中出现。示例 5-18 使用 9.4 版构造重新访问我们的不平衡数组。

示例 5-18：使用多实参 `unnest` 取消不平衡数组的嵌套

```
SELECT * FROM unnest('{blind,mouse}'::text[], '{1,2,3}'::int[]) As f(t,i);

t      | i
-------+---
blind  | 1
mouse  | 2
<NULL> | 3
```

5.5　区间类型

区间数据类型（http://www.postgresql.org/docs/current/interactive/rangetypes.html）是 9.2 版引入的一项新特性，该数据类型可以定义一个值区间。由于该类型的出现，原本需要两个字段才能定义的区间现在仅使用一个字段即可。PostgreSQL 为区间类型提供了很多配套的

运算符和函数，例如判定区间是否重叠，判定某个值是否落在区间内，以及将相邻的若干区间合并为一个完整的区间等。在出现区间类型之前，类似操作只能通过写函数实现，这种操作很繁琐，不仅低效而且很容易出错，并且写出的函数不一定能达到预想的效果，在对于时间类型的操作中尤其如此。在我们自己的项目中，我们在所有需要表示时间范围的表中都用上了区间类型，事实证明效果很好。我们希望你也能分享我们这一成功经验。

有了区间类型后，就不再需要用两个字段来定义一个区间。假设我们希望定义一个大于等于 –2 小于 2 的整数区间，该区间的写法是 `[-2,2)`，左边中括号表示左边是闭区间，即值域包含 –2；右边小括号表示右边是开区间，即值域不包含 2。那么 `[-2,2)` 这个整数区间包含的元素有：–2，–1，0，1。类似地，可以知道以下整数区间所包含的元素。

- 整数区间 `(-2,2]` 含四个元素：`-1`、`0`、`1`、`2`。
- 整数区间 `(-2,2)` 含三个元素：`-1`、`0`、`1`。
- 整数区间 `[-2,2]` 含五个元素：`-2`、`-1`、`0`、`1`、`2`。

5.5.1　离散区间和连续区间

PostgreSQL 对离散区间和连续区间是区别对待的。整数类型或者日期类型的区间是离散区间，因为区间内每一个值都是可以被枚举出来的。数字区间或者时间戳区间就是一个连续区间，因为区间内的值有无限多。

一个离散区间有多种表示方法，比如我们前面提到的 `[-2,2)` 这个例子，它就可以换用多种写法而且每种方式的效果完全一样：`[-2,1]`、`(-3,1]`、`(-3,2)` 和 `[-2,2)`。这四种写法中，PostgreSQL 规定 `[-2,2)` 为规范写法，并不是因为这种写法有什么优势，仅仅是因为统一后有利于运算，不用每次计算时都要先考虑是开区间还是闭区间这件事。PostgreSQL 会自动对所有的离散区间进行规范化，不管是存储还是显示时都会这么做。因此，如果你输入了一个时间区间 `(2014-1-5,2014-2-1]`，那么 PostgreSQL 会自动把它改写为 `[2014-01-06,2014-02-02)`。

5.5.2　原生支持的区间类型

PostgreSQL 原生支持六种区间类型，都是关于数字型和日期时间型。

- `int4range`、`int8range`

  这是整数型离散区间，其定义符合前闭后开的规范化要求。

- `numrange`

  这是连续区间，可以用于描述小数、浮点数、或者双精度数字的区间。

- `daterange`

  这是不带时区信息的日期离散区间。

- tsrange、tstzrange

  这是时间戳（日期加时间）类型的连续区间，秒值部分支持小数。tsrange 不带时区信息，tstzrange 带时区信息。

对于数字类型的区间来说，如果区间的起点值或者终点值未指定，那么 PostgreSQL 会自动为其填入 null 值。理论上讲，你可以将该 null 解释为代表左侧的 -infinity（负无穷）或右侧的 infinity（正无穷）。实际上，你会受限于特定数据类型的最小值和最大值。比如对于 int4range 数据类型来说，区间 (,) 实际上代表的是 [-2147483648,2147483647)。

对于时间类型的区间来说，-infinity 和 infinity 就是有效的上限和下限。

除了系统原生支持的区间类型外，你还可以自定义区间类型，可以设定为离散区间也可以设定为连续区间。

5.5.3 定义区间的方法

任何类型的区间都是由相同数据类型的起点值和终点值外加表示区间开闭的符号 [、]、(、) 构成。如示例 5-19 所示。

示例 5-19：使用类型转换的方法来定义区间

```
SELECT '[2013-01-05,2013-08-13]'::daterange; ❶
SELECT '(2013-01-05,2013-08-13]'::daterange; ❷
SELECT '(0,)'::int8range; ❸
SELECT '(2013-01-05 10:00,2013-08-13 14:00]'::tsrange; ❹

[2013-01-05,2013-08-14)
[2013-01-06,2013-08-14)

[1,)
("2013-01-05 10:00:00","2013-08-13 14:00:00"]
```

❶ 定义了一个从 2013-01-05 到 2013-08-13 的日期型闭区间。请注意此处区间终点的写法是不规范的。

❷ 定义了一个从 2013-01-05 到 2013-08-13 的日期型半开半闭区间。请注意此处的写法是不规范的。

❸ 定义了一个大于 0 小于等于正无穷大的整数区间。请注意此处的写法是不规范的。

❹ 定义了一个从 2013-01-05 10:00 到 2013-08-13 14:00:00 的半开半闭连续区间。



PostgreSQL 中的日期时间类型可以接受 -infinity（负无穷）或者 infinity（正无穷）作为其值。为了与传统写法保持一致，我们建议你还是以前闭后开方式来书写 tsrange 和 tstarange 这类连续区间，即区间起点使用左中括号“[”，区间终点使用右小括号“)”。

区间也可以使用 constructor range 函数来定义，该函数的名称与区间名称是一致的，可以输入两个或者三个实参。示例如下：

```
SELECT daterange('2013-01-05','infinity','[]');
```

第三个实参是区间边界开闭标识符，如果不填则默认为 [)。为清晰起见，我们建议你总是显式指定该实参，因为其默认值不是那种非常显而易见的值，容易被记错。

5.5.4 定义含区间类型字段的表

时间类型区间是很常用的，假设你有一张 employment 表，表中存储了公司聘请雇员的历史记录。你可以像示例 5-20 那样用时间区间来定义一个员工在公司的服务年限，而不需要用起始时间和结束时间两个字段来表示。在本例中，我们给 period 列添加了一个索引以使用我们的区间列加速查询。

示例 5-20：建立一个带有日期区间类型字段的表

```
CREATE TABLE employment (id serial PRIMARY KEY, employee varchar(20), period dater
ange);
CREATE INDEX idx_employment_period ON employment USING gist (period); ❶
INSERT INTO employment (employee, period)
VALUES ('Alex', '[2012-04-24, infinity)'::daterange), ('Sonia', '[2011-04-24,
2012-06-01)'::daterange), ('Leo', '[2012-06-20, 2013-04-20)'::daterange), ('Regi
na', '[2012-06-20, 2013-04-20)'::daterange);
```

❶ 在区间字段上建立一个 GiST 索引。

5.5.5 适用于区间类型的运算符

区间类型上用得最多的两个运算符是重叠运算符（&&）和包含运算符（@>）。要了解区间运算符的完整列表，请参考 PostgreSQL 官方手册中的“区间类型运算符”一节（http://www.postgresql.org/docs/current/interactive/functions-range.html#RANGE-OPERATORS-TABLE）。

1. 重叠运算符

顾名思义，重叠运算符 && 的作用就是判定两个区间是否有重叠部分，如果有则返回 true，否则返回 false。示例 5-21 演示了该运算符的用法，其中还使用了 string_agg 函数将雇员名单列表合并成一个文本字段。

示例 5-21：查询谁与谁曾经同时在公司工作过

```
SELECT e1.employee, string_agg(DISTINCT e2.employee, ', ' ORDER BY e2.employee) As
colleagues
FROM employment As e1 INNER JOIN employment As e2
ON e1.period && e2.period
WHERE e1.employee <> e2.employee
GROUP BY e1.employee;
```

```
employee | colleagues
---------+-------
Alex     | Leo, Regina, Sonia
Leo      | Alex, Regina
Regina   | Alex, Leo
Sonia    | Alex
```

2. 包含与被包含关系运算符

对于包含关系运算符 `@>` 来说，第一个实参是区间，第二个实参是待判定的值。如果第二个实参的值是落在第一个实参的区间内的话，运算符就返回 `true`，否则返回 `false`。示例 5-22 演示了其用法：

示例 5-22：查询当前还在公司工作的雇员名单

```
SELECT employee FROM employment WHERE period @> CURRENT_DATE GROUP BY employee;

employee
----
Alex
```

`<@` 是用于判定被包含关系是否成立的运算符，它的第一个实参是待判定的值，第二个实参是区间，其用法与包含关系运算符完全一致，不再赘述。

5.6 JSON数据类型

JSON 数据类型及其相关操作函数是从 9.2 版开始支持的。JSON 是 Web 开发领域非常流行的一种数据类型，它是 JavaScript 语言中的通用数据交换格式。9.3 版针对 JSON 类型新增了一些功能函数，用以执行读取、编辑以及转换为其他数据类型等操作。有了这些函数以后，PostgreSQL 对于 JSON 的支持力度大大增强。9.4 版引入了 `jsonb` 数据类型，该类型是 JSON 类型的二进制版本，它与 JSON 类型最主要的差别是 JSONB 可以支持索引而 JSON 不能。以下我们将主要介绍 9.3 版中引入的 JSON 处理函数和相关运算符。此外也会介绍如何使用 `jsonb`，如何使用 `jsonb` 与其 `json` 同胞共享的函数，以及如何使用 `jsonb` 所支持的运算符。前述运算符和函数的完整列表可参考 PostgreSQL 官方手册中"JSON 类型的处理函数和运算符"（http://www.postgresql.org/docs/current/interactive/functions-json.html）这一节。

5.6.1 插入JSON数据

要想在表中存储 `json` 数据，只需建一个 `json` 类型的字段即可，语法如下：

```
CREATE TABLE families_j (id serial PRIMARY KEY, profile json);
```

示例 5-23 的语句向表中插入了一条 JSON 记录。PostgreSQL 会自动对插入的 JSON 文本进行格式检查以确保其合法。

示例 5-23：插入一条 JSON 数据记录

```
INSERT INTO families_j (profile) VALUES (
'{"name":"Gomez", "members":[
{"member":{"relation":"padre", "name":"Alex"}},
{"member":{"relation":"madre", "name":"Sonia"}},
{"member":{"relation":"hijo", "name":"Brandon"}},
{"member":{"relation":"hija", "name":"Azaleah"}}
]}');
```



无效的 JSON 字符串是无法转换为 json 类型的，同样也无法将无效的 JSON 字符串存储到某个 json 列中。PostgreSQL 会在后台进行检查以确保 JSON 字符串运行良好，然后才会让 JSON 字符串驻留到数据库中。

5.6.2 查询JSON数据

9.3 版中引入了多种访问 JSON 数据的函数。示例 5-24 使用了 json_extract_path、json_array_elements 以及 json_extract_path_text 这三个函数来读取表中所有家庭成员的信息。

示例 5-24：查询 JSON 数据块中的元素

```
SELECT json_extract_path_text(profile, 'name') As family, ❶ json_ex
tract_path_text( ❷  json_array_elements( ❸   json_extract_path(profile,'mem
bers')  ❹  ), 'member','name' ) As member
FROM families_j;

family |member
-------+------
Gomez  |Alex
Gomez  |Sonia
Gomez  |Brandon
Gomez  |Azaleah
```

❶ 提取出家庭的名称，以文本格式输出。

❷ 提取出家庭成员的名称，以文本格式输出。

❸ 将家庭成员信息数组中的每个元素展开为独立的 JSON 对象。

❹ 获取所有家庭成员的信息列表，作为一个独立的 JSON 对象输出。

运算符 ->> 和 #>> 是 json_extract_path_text 的简写。#>> 取用某个路径数组。示例 5-25 使用这些符号运算符对示例 5-24 进行了重写。

示例 5-25：使用运算符实现与按路径读取函数相同的功能

```
SELECT profile->>'name' As family, json_array_elements((profile->'members')) #>>
'{member,name}'::text[] As member
FROM families_j;
```

json_extract_path 是 json_extract_path_text 的兄弟函数，它对应的运算符是 -> 和 #>。

该函数输出的执行结果是当前 JSON 对象的子对象。如果要把一个复合 JSON 对象（即包含多条记录的 JSON 对象，比如本例中的 members 对象就是一个包含了多条家庭成员信息记录的复合 JSON 对象）剥离出来并传递给别的函数做进一步处理，就需要使用 json_extract_path 函数。

如果你使用的是 9.2 版，情况会稍微有点麻烦，因为该版本中未原生提供能方便地访问 JSON 数据的函数，但可以通过自行编写 PL/V8 函数来实现类似 9.3 版中提供的 JSON 函数的功能。我们在“使用 PLV8 语言来编写 JSON 访问函数”这篇博文（http://www.postgresonline.com/journal/archives/272-Using-PLV8-to-build-JSON-selectors.html）中介绍了如何编写 jQuery 风格的 JSON 数据访问函数。

有若干函数可用于处理 JSON 数组数据，前面我们已经在示例 5-25 中演示了 json_array_elements 这个函数的用法。此外还有一个可以查询数组元素个数的 json_array_length 函数以及可以根据下标引用 JSON 数组元素的运算符 ->。你可以级联使用多个运算符来定位到 JSON 对象内部的某个子对象，如示例 5-26 所示。

示例 5-26：查询 members 对象的子对象

```
SELECT id, json_array_length(profile->'members') As numero, profile->'members'-
>0#>>'{member,name}'::text[] As primero
FROM families_j;

id | numero | primero
---+--------+-----------
1  |    4   | Alex
```

示例 5-26 中使用了 -> 运算符的两种形式。-> 运算符的返回结果永远是一个 json 或者 jsonb 对象，但该运算符的第二个实参要么是一个 text 字段（json_object_field 的简写），要么是一个 integer（json_array_element 的简写）。因此 profile->'members' 会返回 JSON 对象的 members 字段，该字段本身是一个包含多条记录的 JSON 数组。->0 操作提取出了 JSON 对象数组的首个元素。在本例中，->0 得到的是首个家庭成员的信息。#>>'{member,name}'::text[] 就是 json_extract_path_text 操作，得到的结果是首个家庭成员 JSON 对象中按照“member/name”路径寻址到的节点的文本格式的值。通过这个例子你应该可以看出来，这些运算符是可以级联使用的。jsonb 类型也有相同的运算符，不过其对应的函数分别是 jsonb_object_field 和 jsonb_array_element，可以看到就是把函数名中的“json”换成了“jsonb”，其他函数以此类推。



JSON 数组下标是从 0 开始，但 PostgreSQL 的数组下标是从 1 开始。

5.6.3 输出JSON数据

PostgreSQL 除了可以查询库中已有的 JSON 数据外，还支持将别的数据类型转换为 JSON 类型。接下来的例子里面，我们将演示系统内置的 JSON 转换函数的用法，这类函数可以将其他数据类型转换为 JSON 类型。

示例 5-27 中，我们将使用 `row_to_json` 函数将前面示例 5-23 中导入的数据的部分字段转换为 JSON 数据。

示例 5-27：将多条记录转换为单个 JSON 对象（PostgreSQL 9.3 及之后的版本才支持该语句）

```
SELECT row_to_json(f) As x
FROM (SELECT id, profile->>'name' As name FROM families_j) As f;

             x
-------------------------
{"id":1,"name":"Gomez"}
```

如果要将 `families` 表中的所有记录行整体打包转换为一个 JSON 对象，可以使用以下语法（PostgreSQL 9.2 及之后的版本均支持该语法）：

```
SELECT row_to_json(f) FROM families_j As f;
```

“查询时将一行记录作为单个字段输出”这种功能只有 PostgreSQL 才支持。该功能对于创建复合 JSON 对象特别有用。我们将在 7.2.10 节中深入讨论此特性，并且在示例 7-16 中演示如何使用 `array_agg` 和 `array_to_json` 函数将多条记录转换为一个 JSON 对象后输出。9.3 版新增了对 `json_agg` 函数的支持，我们将在示例 7-17 中演示此函数的用法。

5.6.4 JSON类型的二进制版本：jsonb

PostgreSQL 9.4 版中引入了新的 `jsonb` 数据类型。从运算符的角度看，`jsonb` 有若干 `json` 类型不支持的运算符，此外二者的运算符完全相同。从处理函数角度看，二者适用的处理函数一一对应，仅在命名上略有差别，一个以“json”开头，一个以“jsonb”开头。`jsonb` 数据类型和 `json` 数据类型的关键区别如下所示。

- `json` 是以原始文本格式存储的，而 `jsonb` 存储的是原始文本解析以后生成的二进制数据结构，该二进制结构中不再保存原始文本中的空格，存储下来的数字的形式也发生一定的变化，并且对其内部记录属性值进行了排序。例如，文本中的 `e-5` 这种数字会被转换为对应的小数存储。
- `jsonb` 不允许其内部记录的键值重复，如果出现重复则会从中自动选择一条，其余的重复记录会被丢弃，但 `json` 类型中记录键值重复是允许的。Michael Paquier 的“利用 `jsonb` 类型不允许键值重复的特性来管理 `jsonb` 数据”博文（http://michael.otacoo.com/postgresql-2/manipulating-jsonb-data-with-key-unique/）中演示了若干例子。

- jsonb 的性能远好于 json。因为 jsonb 类型在处理过程中不需要再进行文本解析。
- jsonb 类型由于是解析过的二进制结构，因此 jsonb 类型的字段上可以直接建立 GIN 索引（该类索引在 6.3 节中有相关介绍），但 json 类型字段上却只能建立函数索引，因为只有通过函数才能从 JSON 的字符串中提取出具体字段值。

为了说明以上概念，我们另外新建一张与前面 families_j 结构类似的 families_b 表，只不过这次用的是 jsonb 类型：

```
CREATE TABLE families_b (id serial PRIMARY KEY, profile jsonb);
```

重复执行示例 5-23 的步骤，往新表中插入记录。

目前为止还没体现出 JSON 和 JSONB 的差别，但在对两张表分别执行查询时就能看出来了。为了让 JSONB 类型的二进制字段值能够显示，PostgreSQL 会自动将其转换为规范化的文本表示形式，如示例 5-28 所示。

示例 5-28：JSONB 与 JSON 类型输出格式对比

```
SELECT profile As b FROM families_b WHERE id = 1; ❶
SELECT profile As j FROM families_j WHERE id = 1;❷

b
--------------------------------------------------------------------------------
 {"name": "Gomez", "members": [{"member": {"name": "Alex", "relation": "padre"}}
, {"member": {"name": "Sonia", "relation": "madre"}}, {"member": {"name": "Brand
on", "relation": "hijo"}}, {"member": {"name": "Azaleah", "relation": "hija"}}]}

j
---------------------------------------------------------------------------
{"name":"Gomez","members":[{"member":{"relation":"padre", "name":"Alex"}},
{"member":{"relation":"madre", "name":"Sonia"}},
{"member":{"relation":"hijo", "name":"Brandon"}},
{"member":{"relation":"hija", "name":"Azaleah"}}]}
```

❶ 可以看出，jsonb 类型的输出是对输入的内容进行了重新格式化并删掉了输入时文本中的空格，此外 relation 和 name 这两个属性字段的显示顺序与输入时的顺序相比发生了翻转。

❷ json 类型的输出保持了输入时的原样，包括原文本中的空格以及属性字段的顺序。

jsonb 与 json 的处理函数一一对应，但函数名略有不同；jsonb 支持的运算符集合是 json 支持的运算符集合的超集。例如 json 适用的 json_extract_path_text 和 json_each 函数对应于 jsonb 适用的 jsonb_extract_path_text 和 jsonb_each 函数。除了 jsonb 特有的那几个运算符以外，二者的运算符完全相同，因此如果要想把示例 5-25 和示例 5-26 中的语句改造为适用 jsonb 类型，只需把表名替换一下（families_j 改为 families_b），然后把 json_array_length 替换为 jsonb_array_length 即可。

jsonb 比 json 多支持的运算符有以下几个：等值运算符（=）、包含关系运算符（@>）、被包含关系运算符（<@）、键值已存在运算符（?）、一组键值中是否有任意一个已存在运算符（?|）、一组键值中的每一个是否均已存在运算符（?&）。

假设我们要列出所有包含姓名为“Alex”的家庭成员的家庭，就可以使用包含关系判定运算符，如示例 5-29 所示。

示例 5-29：JSONB 包含关系运算符的使用

```
SELECT profile->>'name' As family
FROM families_b
WHERE profile @> '{"members":[{"member":{"name":"Alex"} }]}';

family
-----
Gomez
```

如果在 jsonb 列上建了 GIN 索引，那么前述这几个运算符的操作速度是极快的：

```
CREATE INDEX idx_familes_jb_profile_gin ON families_b USING gin (profile);
```

我们演示用的这些表都很小，因此规划器可能会选择走全表扫描而不是走索引查询，但如果有更多的记录，示例 5-29 中这种语句是一定会用上索引的。

5.7 XML数据类型

XML 和 JSON 这两种数据类型都属于非规范化数据，在关系型数据库中存储这类数据其实是有争议的。然而，所有的高级关系型数据库（比如 IBM DB2、Oracle、SQL Server）中都支持 XML 数据类型。作为最先进的开源关系型数据库，PostgreSQL 自然也会支持 XML 数据类型，并且还提供了大量 XML 操作函数。我们发表过很多关于在 PostgreSQL 中使用 XML 数据类型的技术文章（http://www.postgresonline.com/journal/index.php?/plugin/tag/xml）。PostgreSQL 原生支持创建、管理和解析 XML 数据的函数，详细列表可参见 PostgreSQL 官方手册中“XML 函数”这一节（http://www.postgresql.org/docs/current/interactive/functions-xml.html）。与 jsonb 数据类型不一样，目前没有哪种索引类型支持直接对 XML 数据类型进行索引，因此只能使用函数索引对其一部分数据进行索引，这一点与 json 是相同的。

5.7.1 插入XML数据

在往一个 xml 数据类型的列中插入数据时，PostgreSQL 会自动判定并确保只有格式合法的 XML 才会创建成功。text 类型字段中也可以存入一段 XML 文本，但是存入时不会进行格式合法性判断，这一点是 text 与 xml 类型的区别。不过请注意，即使 XML 文本的内容中附带了 DTD 或者 XSD 的格式描述，PostgreSQL 也不会按照这些格式要求来对 XML 的格

式进行验证。为了梳理一下构成有效 XML 的要素，示例 5-30 向你展示了通过将某个列声明为 `xml` 并照常将数据插入到该列中，你如何将 XML 数据追加到表中。

示例 5-30：插入 XML 字段记录

```
CREATE TABLE families (id serial PRIMARY KEY, profile xml);
INSERT INTO families(profile)
VALUES(
    '<family name="Gomez">
        <member><relation>padre</relation><name>Alex</name></member>
        <member><relation>madre</relation><name>Sonia</name></member>
        <member><relation>hijo</relation><name>Brandon</name></member>
        <member><relation>hija</relation><name>Azaleah</name></member>
        </family>');
```

XML 数据的格式是千变万化的，你可以为 XML 字段设置一个 check 约束以确保输入的 XML 数据都符合某种格式（如需了解 check 约束的详细信息，请参考 6.2.3 节的内容）。示例 5-31 中创建了一个 check 约束，该约束要求输入的 XML 数据中的 `family` 节点下都有一个 `relation` 节点。`'/family/member/relation'` 是 XPath 语法，XPath 是一种能够在 xml 树状结构中定位到指定元素的语法。

示例 5-31：确保所有 XML 字段记录中都有至少一个 `member` 节点和一个 `relation` 节点

```
ALTER TABLE families ADD CONSTRAINT chk_has_relation
CHECK (xpath_exists('/family/member/relation', profile));
```

如果我们试图插入这样一条记录：

```
INSERT INTO families (profile) VALUES('<family name="HsuObe"></family>');
```

我们会看到这样的报错信息：`ERROR: new row for relation "families" violates check constraint "chk_has_relation"`（错误：试图插入“families”表中的新记录违反了约束“chk_has_relation”的要求）。

如果需要基于 DTD 或者 XSD 对 XML 数据进行格式检查，你需要自行编写格式检查函数，然后将此函数放到 check 约束中调用。PostgreSQL 目前还没有原生支持基于 DTD 或者 XSD 的格式检查。

5.7.2　查询XML数据

查询 XML 数据时，`xpath` 函数会发挥重要作用。该函数的第一个实参是一个 XPath 查询表达式，第二个实参是一个 `xml` 对象。查询结果是 XPath 查询语句所要查找的 XML 元素的列表。示例 5-32 中查询出了所有的家庭成员，查询中同时使用了 `xpath` 和 `unnest` 函数，其中 `unnest` 函数用于将数组转换成结果集。这样我们就把一段 XML 中的零碎信息提取出来并转换成了文本。

示例 5-32：查询 XML 字段

```
SELECT family,
    (xpath('/member/relation/text()', f))[1]::text As relation,
    (xpath('/member/name/text()', f))[1]::text As mem_name ❶
FROM (SELECT (xpath('/family/@name', profile))[1]::text As family, ❷
 unnest(xpath('/family/member', profile)
 ) As f FROM families) x; ❸

 family | relation | mem_name
--------+----------+----------
 Gomez  | padre    | Alex
 Gomez  | madre    | Sonia
 Gomez  | hijo     | Brandon
 Gomez  | hija     | Azaleah
```

❶ 获取每个 `member` 元素的 `relation` 标签和 `name` 标签中包含的文本元素。此处的语法中必须加数组下标，因为 `xpath` 语法返回的查询结果是数组类型的，即使返回的数组中只有一个元素也得加下标才能访问。

❷ 获取 `family` 根节点的 `name` 属性值。访问属性值的语法为 `@attribute_name`。

❸ 从原始的 XML 文本块中提取出所有的 `member` 节点的内容，每个 `member` 节点的内容格式为 `<member>`、`<relation>`、`</relation>`、`<name>`、`</name>` 和 `</member>`。`xpath` 的斜杠语法表示要获取当前指定节点的子节点的内容。例如，`xpath('/family/member', 'profile')` 将以数组形式返回 `profile` 字段中 `family` 节点下所有 `member` 子节点的内容。`xpath('/family/@name', 'profile')` 返回的是 `family` 节点的 `name` 属性的值。默认情况下，`xpath` 返回的是包含前后标签部分的完整节点内容，加了 `text()` 以后，返回的就是该节点中包含的文本的内容。

5.8 自定义数据类型和复合数据类型

本节将为你介绍如何定义和使用自定义数据类型。`composite`（也称为 `record`，`row`）常用于构建需要转为自定义数据类型的对象或者是作为需要返回多个字段的函数的返回值类型定义。

5.8.1 所有表都有一个对应的自定义数据类型

PostgreSQL 在建表时会自动创建一个与表结构完全相同的自定义数据类型，而且这种类型与其他的数据类型在使用上毫无区别。可以在建表时指定某字段为表类型或者表数组类型，也就是说可以把一张表的字段定义为另一张表。这种将表层层嵌套的用法与“turducken”很像（turducken 即“特大啃”，是 turkey-duck-chicken 的简称，一种美食新吃法：将无骨鸡填到无骨鸭的肚子里，然后再将填了无骨鸡的无骨鸭填到无骨火鸡的肚子里，这种食物很好地体现了多层嵌套的关系）。在示例 5-33 中，我们将用“特大啃”的例子来演示。

示例 5-33：为“特大啃”建嵌套表

```
CREATE TABLE chickens (id integer PRIMARY KEY);
CREATE TABLE ducks (id integer PRIMARY KEY, chickens chickens[]);
CREATE TABLE turkeys (id integer PRIMARY KEY, ducks ducks[]);

INSERT INTO ducks VALUES (1, ARRAY[ROW(1)::chickens, ROW(1)::chickens]);
INSERT INTO turkeys VALUES (1, array(SELECT d FROM ducks d));
```

上面我们直接在 ducks 表的一条记录的 chickens 字段中插入了两条 chickens 记录，这种情况下这两条记录的构造不受 chickens 表定义的约束，因此即使它们的主键重复也没关系。我们生成了两条 chickens 记录，填入 ducks 表中，然后将这条 ducks 记录填入到 turkeys 表中，这个过程相当于把两只 chicken 塞入一只 duck，然后再把这只 duck 塞入一只 turkey，跟制作“特大啃”的过程是完全一样的。

最后我们看一下得到的 turkeys 记录是什么样子的：

```
SELECT * FROM turkeys;

output
------------------------
id |         ducks
---+----------------------
 1 | {"(1,\"{(1),(1)}\")"}
```

嵌套表中内嵌的表记录是可以进行修改的。例如，我们要对第一个 turkey 内嵌的第二个 chicken 进行修改，那么可以执行如下操作：

```
UPDATE turkeys SET ducks[1].chickens[2] = ROW(3)::chickens
WHERE id = 1 RETURNING *;

output
--------------
id |         ducks
---+----------------------
 1 | {"(1,\"{(1),(3)}\")"}
```

我们使用 RETURNING 子句来返回本次更新操作涉及的所有记录，我们将在 7.2.9 节中介绍 RETURNING 子句的用法。

PostgreSQL 内部维护着数据库对象之间的依赖关系。前述 ducks 表的 chickens 字段依赖于 chickens 表，turkeys 表的 ducks 记录依赖于 ducks 表。要想删除 chickens 表有两个方法，要么在 drop 语句中带上 CASCADE 关键字，要么先删除 ducks 表中的 chickens 字段。如果使用前一个方法，那么 ducks 表的 chickens 字段会被自动删除，而且此过程中无告警信息。相应地，turkeys 表的 ducks 字段的定义也将自动跟着改变。

5.8.2 构建自定义数据类型

尽管仅仅通过建表就可以轻松创建复合数据类型，但有时候我们仍会需要从头开始构建自己的数据类型。例如，使用以下语句可以构建一个复杂数字数据类型：

```
CREATE TYPE complex_number AS (r double precision, i double precision);
```

可以将此类型作为字段类型定义使用：

```
CREATE TABLE circuits (circuit_id serial PRIMARY KEY, ac_volt complex_number);
```

可以使用如下语法对这个表进行查询：

```
SELECT circuit_id, (ac_volt).* FROM circuits;
```

或者这种语法也可以：

```
SELECT circuit_id, (ac_volt).r, (ac_volt).i FROM circuits;
```



你可能会问：上面语句中 `ac_volt` 外面为什么要加括号？如果不加的话，PostgreSQL 会报错说 `FROM` 子句中找不到 `ac_volt` 这张表。根据这一点就很好理解了，加括号的原因是为了不让 PostgreSQL 将其理解为表名。

5.8.3 为自定义数据类型构建运算符和函数

在构建自定义数据类型后，你自然就会需要为其创建相应的函数和运算符。我们接下来将演示如何为 `complex_number` 类型创建一个 + 运算符，而创建处理函数的方法将放在本书后面的第 8 章中进行介绍。我们在前面已经介绍过，每个运算符都有一个底层实现函数，该函数需要一个或者两个实参，运算符就是这个函数的符号化别名。在 PostgreSQL 官方手册的“创建运算符”这一节（http://www.postgresql.org/docs/current/interactive/sql-createoperator.html）中你可以看到系统允许使用哪些字符来定义新的运算符。

运算符不仅仅是其底层实现函数的别名，它还可以提供一些可以帮助规划器更好工作的优化信息，规划器借助这些信息可以判定如何使用索引，如何以最低的成本访问数据，以及哪些运算符表达式是等价的。这些信息的完整列表以及每一类信息的具体作用可以参考官方手册中“运算符的优化信息”这一节的内容（http://www.postgresql.org/docs/current/interactive/xoper-optimization.html）。

创建运算符的第一步是创建其底层实现函数，如示例 5-34 所示。

示例 5-34：为 `complex_number` 创建底层实现函数

```
CREATE OR REPLACE FUNCTION add(complex_number, complex_number) RETURNS complex_num
ber AS
```

```
$$
 SELECT ( (COALESCE(($1).r,0) + COALESCE(($2).r,0)),
    (COALESCE(($1).i,0) + COALESCE(($2).i,0)) )::complex_number;
$$
language  sql;
```

接下来要创建一个运算符来代表此函数，如示例 5-35 所示。

示例 5-35：为 complex_number 类型定义 + 运算符

```
CREATE OPERATOR +(
  PROCEDURE = add,
  LEFTARG = complex_number,
  RIGHTARG = complex_number,
  COMMUTATOR = +);
```

然后我们测试一下这个新的 + 运算符：

```
SELECT (1,2)::complex_number + (3,-10)::complex_number;
```

输出结果是 (4,-8)。

虽然我们在此处没有举例说明，但你可以对函数和运算符进行重载，以使其可以接受多种不同类型的输入。例如，你可以创建一个支持 complex_number 和 integer 相加的 add 函数和相应的 + 计算符，这就实现了对原逻辑的扩展。

支持自定义数据类型和运算符让 PostgreSQL 从机制上具有了自我演进的能力，开源社区无数开发人员利用此能力为 PostgreSQL 平台添砖加瓦，随着这个开发平台的羽翼日渐丰满，我们离“一切皆以表驱动”的理想境界也越来越近。

第 6 章

表、约束和索引

表是关系型数据库存储体系的基本单元。设计好结构化的表并且定义表与表之间的关联关系是关系型数据库的核心设计思想。在 PostgreSQL 中，约束定义了表与表之间的关系。与以堆结构存储的记录相比，表的优势就在于有索引。你会在很多书的末尾看到词汇索引表，也会在写字楼入口大堂处看到每层楼的租户名单，表索引的作用与它们是类似的，即在表中快速查找到目标数据的位置，这样你就可以不用每次查询时都扫描整张表的内容。

本章中，我们将介绍建表和插入记录的语法。然后将介绍约束的用法，约束可以保证数据库的记录不会违反我们制定的规则。最后我们将向你展示如何为表创建索引来加速查询。在表上建索引是需要经过深思熟虑的，因为一个错误的索引会导致查询效果比全表扫描还慢，也就是说建了还不如不建。并不是所有的索引都是“生来平等”的，数据库领域的算法专家们为不同的数据类型设计出了不同类型的索引，目的是将查询的速度提升到极致。

6.1 表

除了普通的表以外，PostgreSQL 还提供了许多不常见的表，具体包括：临时表、无日志表、继承表、基于复合类型的表以及外部表（我们将在第 10 章中介绍外部表）。

6.1.1 基本的建表操作

示例 6-1 演示了建表语法，在所有支持 SQL 的数据库中建表语法都是类似的。

示例 6-1：基本的建表操作

```
CREATE TABLE logs ( log_id serial PRIMARY KEY, ❶ user_name varchar(50), ❷ descrip
```

```
tion text, ❸ log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp);
❹
CREATE INDEX idx_logs_log_ts ON logs USING btree (log_ts);
```

❶ `serial` 数据类型是一种自增长的数字类型。建表时如果有一个 `serial` 类型的字段，那么系统会自动在 schema 中同时创建一个对应的序列号生成器。`serial` 类型字段的值是一个整型数字，它会自动被赋值为序列号生成器的下一个值。每张表一般来说只会有一个 `serial` 字段，且一般是用作主键。

❷ `varchar` 是 `character varying`（可变长字符串）的简写，其定义与其他数据库产品中的定义是类似的。你可以不为 `varchar` 字段设定最大长度值，此时它与 `text` 类型几乎是一样的。

❸ `text` 是一种不定长度的字符串，无最大长度限制。

❹ `timestamp with time zone`（可简写为 `timestamptz`）是一种表示日期和时间的类型，总是以国际标准时间（UTC）格式存储。该类型在显示时总是以服务器当前所在时区为基准进行显示，当然你也可以要求使用指定的时区进行显示。更多关于此类型的讨论，请参考 5.3.1 节。

6.1.2　继承表

PostgreSQL 是唯一提供表继承功能的数据库。如果创建一张表（子表）时指定为继承自另一张表（父表），则建好的子表除了含有自己的字段外还会含有父表的所有字段。PostgreSQL 会记录下这个继承关系，这样一旦父表的结构发生了变化，子表的结构也会自动跟着变化。这种父子继承结构的表可以完美地适用于需要数据分区的场景。当查询父表时，PostgreSQL 会自动把子表的记录也取出来。值得注意的是，并不是所有父表的特征都会被子表继承下来，比如主表的主键约束、唯一性约束以及索引就不会被继承。Check 约束会被继承，但子表还可以另建自己的 check 约束（详见示例 6-2）。

示例 6-2：创建继承表

```
CREATE TABLE logs_2011 (PRIMARY KEY(log_id)) INHERITS(logs);
CREATE INDEX idx_logs_2011_log_ts ON logs USING btree(log_ts);
ALTER TABLE logs_2011 ADD CONSTRAINT chk_y2011
 CHECK (log_ts >= '2011-1-1'::timestamptz
   AND log_ts < '2012-1-1'::timestamptz ); ❶
```

❶ 我们定义了一个 check 约束来限制只能录入 2011 年的数据。该 check 约束告诉查询规划器在查询父表时跳过条件不满足的子表。

6.1.3　无日志表

对于发生磁盘故障或者系统崩溃后可以被重建的临时数据来说，其操作速度比可靠性更重要。PostgreSQL 从 9.1 版开始支持 `UNLOGGED` 修饰符，使用该修饰符可以创建无日志的表，

如示例 6-3 所示。系统不会为这种表记录任何事务日志（业界一般也称为 WAL 日志，即 write-ahead log）。如果你的服务器经历了一次掉电重启，那么无日志表中的数据会在事务回滚过程中被全部清除掉。你可以从 Depesz 站点的“等待 9.1 版的无日志表”这篇博文（http://www.depesz.com/2011/01/03/waiting-for-9-1-unlogged-tables/）中看到更多示例和使用注意事项。

另外在 pg_dump 中还有一个选项可以设置在备份时跳过无日志表。

示例 6-3：创建无日志表

```
CREATE UNLOGGED TABLE web_sessions ( session_id text PRIMARY KEY, add_ts time
stamptz, upd_ts timestamptz, session_state xml);
```

无日志表的一大优势就是对其写入数据要远远快于往普通表中写数据。按照我们的经验，一般要快大约 15 倍。请牢记使用无日志表的缺点。

- 如果数据库服务器崩溃，PostgreSQL 将截断所有无日志表（截断的意思是擦除所有行）。
- 无日志表不支持 GiST 索引（6.3.1 节会讨论此索引类型），因此它就不适用于依赖 GiST 索引的数据类型。

但无日志表上可以建常用的 B- 树索引和 GIN 索引。

6.1.4 TYPE OF

PostgreSQL 在创建一张表时，会自动在后台创建一个结构完全相同的复合数据类型，反之则不是这样。在 9.0 版中，你可以使用一个复合数据类型来作为建表的模板。以下我们将演示该功能，首先创建一个复合数据类型。

```
CREATE TYPE basic_user AS (user_name varchar(50), pwd varchar(10));
```

然后我们可以使用 OF 语法来创建一张表，表结构就是该复合类型，如示例 6-4 所示。

示例 6-4：以复合数据类型为模板来创建一张表

```
CREATE TABLE super_users OF basic_user (CONSTRAINT pk_su PRIMARY KEY (user_name));
```

当基于数据类型来创建表时，你不能指定表字段的定义，一切以数据类型本身的定义为准。然而，为复合数据类型新增或者移除字段时，PostgreSQL 会自动修改相应的表结构。这种机制的优点是，如果你的系统中有很多结构相同的表，而你可能会需要同时对所有表结构进行相同的修改，那么此时只需要修改此基础数据类型即可，这一点与表继承机制很相似。

例如我们需要为示例 6-4 中定义的 super_users 表增加一个电话号码字段，那么只需要修改建表时所使用的数据类型即可。

```
ALTER TYPE basic_user ADD ATTRIBUTE phone varchar(10) CASCADE;
```

通常，如果表依赖于某个类型，那么你就不能更改该类型的定义。CASCADE 修饰符凌驾于此限制之上，对所有相关表应用相同的更改。

6.2 约束机制

PostgreSQL 的约束机制是我们所接触过的数据库中最先进的（同时也是最复杂的）。用户可以在创建约束时定制其各方面的属性，包括约束的名称、如何处理现有数据、级联生效条件选项、如何执行匹配算法、使用哪些索引以及在何种情况下约束可以不生效等。关于完整的约束规则，我们建议你查阅 PostgreSQL 官方手册（http://www.postgresql.org/docs/current/interactive/sql-set-constraints.html）中的相关内容。虽然约束机制中有大量的选项可供定制，但一般来说没那么复杂，采用默认选项就够了。我们将从几个关系型数据库领域耳熟能详的概念开始为你介绍，包括：外键约束、唯一性约束以及 check 约束。然后我们再介绍从 PostgreSQL 9.0 版开始引入的排他性约束。



主键约束和字段唯一性约束在表级范围内不允许出现重名。一般的做法是把表名和字段名加入到约束的名称中，这样就不会重复。为简洁起见，我们下面的例子中可能不会遵循这种做法。

6.2.1 外键约束

与大多数支持引用完整性的数据库一样，PostgreSQL 遵循与其相同的约定。你可以指定级联更新和删除规则以避免出现讨厌的孤立记录。下面我们将在示例 6-5 中展示如何添加外键约束。

示例 6-5：建立外键约束和相应的索引

```
set search_path=census, public;
ALTER TABLE facts ADD CONSTRAINT fk_facts_1 FOREIGN KEY (fact_type_id)
REFERENCES lu_fact_types (fact_type_id) ❶
ON UPDATE CASCADE ON DELETE RESTRICT; ❷
CREATE INDEX fki_facts_1 ON facts (fact_type_id); ❸
```

❶ 我们在 facts 表和 lu_fact_types 表之间定义了一个外键约束关系。有了这个约束以后，如果主表 lu_fact_types 中不存在某 fact_type_id 的记录，那么从表 fact 中就不能插入该 fact_type_id 的记录。

❷ 我们定义了一个级联规则，实现了以下功能：（1）如果主表 lu_fact_type 的 fact_type_id 字段值发生了变化，那么从表 fact 中相应记录的 fact_type_id 字段值会自动进行相应修改，以维持外键引用关系不变；（2）如果从表 fact 中还存在某 fact_type_id 字段值的记录，那么主表 lu_fact_type 中相同 fact_type_id 字段值的记录就不允许

被删除。`ON DELETE RESTRICT` 是默认行为模式，也就是说这个子句不加也可以，但我们建议为了清晰起见最好是加上。

❸ PostgreSQL 在建立主键约束和唯一性约束时，会自动为相应字段建立索引，但在建立外键约束时却不会，这一点需要注意。你需要为外键字段手动建立索引以加快关联引用时的查询速度。

6.2.2 唯一性约束

主键字段的值是唯一的，但每张表只能定义一个主键，因此如果你需要保证别的字段值唯一，那么必须在该字段上建立唯一性约束或者说唯一索引。建立唯一性约束时会自动在后台创建一个相应的唯一索引。与主键字段类似，建立了唯一性约束的字段不允许为空，并且可以作为外键字段被别的表引用。不过请注意：建了唯一索引但却没有唯一性约束的字段是可以输入空值的。下面的例子演示了如何建一个唯一索引。

```
ALTER TABLE logs_2011 ADD CONSTRAINT uq UNIQUE (user_name,log_ts);
```

你可能经常会遇到仅需要保证表中部分记录行唯一的情况，PostgreSQL 不支持带筛选条件的唯一性约束，但你可以通过使用唯一性的部分索引来达到相同目的。详情请参见 6.3.4 节。

6.2.3 check约束

check 约束能够对表的一个或者多个字段加上一个条件，表中每一行记录必须满足此条件。查询规划器也会利用 check 约束来优化执行速度，比如有些查询附带的条件与待查询表的 check 约束无交集，那么规划器会立即认定该查询未命中目标并返回。示例 6-2 中就有一个 check 约束，该约束可以告诉规划器不要试图查找不符合约束条件的记录。check 约束支持基于函数和布尔表达式的条件，因此你可以发挥创意编写出一个非常复杂的约束条件来。例如，以下 check 约束可以限制 `logs` 表中所有用户名必须都小写：

```
ALTER TABLE logs ADD CONSTRAINT chk CHECK(user_name = lower(user_name));
```

特别值得注意的一点是，当表间存在继承关系时，子表会继承父表的 check 约束，但主键、外键、唯一性这三种约束却不会继承。

6.2.4 排他性约束

传统的唯一性约束在比较算法中仅使用了“等于”运算符，即保证了指定字段的值在本表的任意两行记录中都不相等，而 9.0 版中引入的排他性约束机制拓展了唯一性比较算法机制，可以使用更多的运算符来进行比较运算，该类约束特别适用于解决有关时间安排的问题。

PostgreSQL 9.2 版中引入了区间数据类型，该类型特别适合使用排他性约束。你可以在 depesz 站点的“等待 9.2 版支持区间数据类型”这篇博文（http://www.depesz.com/2011/11/07/waiting-for-9-2-range-data-types/）中找到在区间数据类型上使用排他性约束的非常好的例子。

排他性约束一般是基于 GiST 类型的索引来实现，使用基于 B- 树算法的 GiST 多列复合索引也是可以的。不过需要先安装 btree_gist 扩展包才能建立这种索引。多列排他性约束的一个经典应用场景就是用于安排资源。

以下是一个使用排他约束的例子。假设你的办公场所有固定数量的会议室，各项目组在使用会议室前必须预定。示例 6-6 演示了如何避免发生预定冲突。该示例中使用了（&&）运算符来判定时间区段是否重叠，还使用了（=）运算符来判定会议室房间号是否重复，请注意观察和思考此用法。

示例 6-6：防止会议室预定冲突

```
CREATE TABLE schedules(id serial primary key, room smallint, time_slot tstzrange);
ALTER TABLE schedules ADD CONSTRAINT ex_schedules
EXCLUDE USING gist (room WITH =, time_slot WITH &&);
```

同唯一性约束一样，PostgreSQL 会自动为排他性约束中涉及的字段建立索引。

6.3 索引

PostgreSQL 的索引机制功能强大、特性丰富，仅仅索引部分的内容已足够写一本大部头的书。本书写作之时，PostgreSQL 已支持至少四种类型的索引。如果你觉得还不够，PostgreSQL 还允许你为这几种索引类型自定义新的索引运算符和修饰符以作为其功能补充。如果这样还不能满足你的要求，你可以创建自己的索引类型。

PostgreSQL 还支持在同一张表中混合搭配不同的索引类型，且预计规划器将综合考虑所有的索引。例如，在一个字段上建立 B- 树索引，在旁边的字段上建立 GiST 索引，查询时两个索引都可以被用上。要更深入了解规划器对索引的选用机制，请参考 PostgreSQL 官方手册中“位图索引扫描策略”这一节（http://www.postgresql.org/docs/current/interactive/indexes-bitmap-scans.html）的内容。



同一张表上的索引名不允许重复。

6.3.1 PostgreSQL原生支持的索引类型

要想利用好 PostgreSQL 的索引能力，我们需要先了解其支持的不同的索引类型以及它们各自适用于什么场景。索引方法包括以下几种。

- B-树索引

  B- 树是一种关系型数据库中常见的通用索引类型。如果你对别的索引类型不感兴趣，那么一般使用 B- 树索引就可以了。有的场景下 PostgreSQL 会自动创建索引（比如创建主键约束或者唯一性约束时），那么创建出来的索引就是 B- 树类型的；如果你自己创建索引时未指定索引类型，那么默认也会创建 B- 树类型的索引。主键约束和唯一性约束唯一支持的后台索引就是 B- 树索引。

- GiST索引

  GiST 的全称是 Generalized Search Tree，意即通用搜索树。它主要的适用场景包括全文搜索以及空间数据、科学数据、非结构化数据和层次化数据的搜索。该类索引不能用于保障字段的唯一性，也就是说建立了该类型索引的字段上可插入重复值，但如果把该类索引用于排他性约束就可以实现唯一性保障。GiST 是一种有损索引，也就是说它不存储被索引字段的值，而仅仅存储字段值的一个取样，这种取样是失真的，就像把一个盒子变成了一个多边形。这就意味着需要一个额外的查找步骤以获得真正记录的值。

- GIN索引

  GIN 的全称是 Generalized Inverted Index（GIN），即通用逆序索引。它主要适用于 PostgreSQL 内置的全文搜索引擎以及 `jsonb` 数据类型。其他有一些扩展包比如 hstore 和 pg_trgm 也会使用这种索引。GIN 其实是从 GiST 派生出来的一种索引类型，但它是无损的，也就是说索引中会包含有被索引字段的值。如果你需要查询的字段都已被索引，那么只读取索引即可获取查询结果，这种情况下 GIN 的查询速度是快于 GiST 的。然而，由于 GIN 比 GiST 在更新操作时要多出一个字段值复制动作，因此此时是 GiST 索引更快一些。另外，GIN 的索引树内部每一个索引行的长度是有限制的，所以它不能用于对 hstore 文档或者 text 等大对象类型进行索引。如果你需要把一个 600 页的手册内容存入一张表的某个字段，那么绝对不要在该字段上建 GIN 类型的索引。在“等待更快的 LIKE/ILIKE 运算符”（http://www.depesz.com/2011/02/19/waiting-for-9-1-faster-likeilike/）这篇博文中有一个关于 GIN 用法的非常好的例子可供你参考。在 9.3 版中，用于实现字符串模糊匹配和相似度查询的 `pg_trgm` 扩展包中做了一个功能强化：支持正则表达式条件查询时用上 GIN 索引，这大大增加了 `pg_trgm` 的适用场景。

- SP-GiST索引

  SP-GiST 是指基于空间分区树（Space-Partitioning Trees）算法的 GiST 索引。该类型的索引从 9.2 版开始引入，与 GiST 索引适用领域相同，但对于某些特定领域的数据算法，

其效率会更高一些。PostgreSQL 的 `point` 和 `box` 等原生几何类型以及 `text` 类型是最先支持该类索引的数据类型。从 9.3 版开始，区间类型也开始支持此类型的索引。PostGIS 扩展包也有计划要在近期开始用上此类索引以提升性能。

- *哈希索引*

  哈希索引在 GiST 和 GIN 索引出现前就已经得到了广泛使用。业界普遍认为 GiST 和 GIN 索引在性能和事务安全性方面要胜过哈希索引。请注意：事务日志中不会记录哈希索引的变化，那么在流式复制环境中就不能使用哈希索引，否则会导致修改无法被同步。PostgreSQL 已将哈希索引列为不推荐使用状态。在别的数据库中你可能仍会见到该索引类型，但在 PostgreSQL 中最好避免使用它。

- *基于B-树算法的GiST和GIN索引*

  如果你想了解 PostgreSQL 除了默认特性以外还有哪些功能，不管是出于业务需要或者是仅仅出于好奇心，都可以从了解基于 B- 树算法的 GiST 和 GIN 索引开始。这两种复合索引非常具有代表性，而且二者都是以扩展包形式存在的，因此也比较便于学习和研究。这两类混合算法索引的优势在于，它们一方面能够支持 GiST 和 GIN 索引特有的运算符，同时又具有 B- 树索引对于“等于”运算符的良好支持。有时我们会需要建立这样的多列复合索引：索引字段中既有像 `character varying` 或 `number` 这样的数据类型，又有层次化的 `ltree` 类型或者用于全文搜索的 `vector` 类型。前两种数据类型一般会使用“等于”运算符来进行操作，后两种一般使用 GIN/GiST 索引提供的运算符进行操作。此时你会发现要建立这种索引必须使用基于 B- 树算法的 GiST 或者基于 B- 树算法的 GIN。

6.3.2 运算符类

其实我们非常希望能够跳过这部分关于“运算符类”的内容，因为很多幸运的人即使不知道“运算符类”是什么以及它与索引有什么关系也能毫无障碍地使用索引。但如果你运气没这么好，也就是说你的应用场景需要你了解这些内容，那么就得对运算符类好好研究了，否则就会一直被这个问题困扰：“为什么规划器没用上我的索引？”

各种数据类型均有其自身特点，因此适用的索引类型不同，会用到的比较运算符也不同。例如，对于基于区间类型（range）的索引来说，最常用的运算符是重叠运算符（`&&`），然而该运算符对于快速本文搜索领域来说却毫无意义。对于中文这类表意文字来说，建立的索引基本上不会用到“不等于”运算符；而对英文这类表音文字建立索引时，字母 A 到 Z 的排序操作是不可或缺的。

基于以上特点，PostgreSQL 把一类应用领域相近的运算符以及这些运算符适用的数据类型组合在一起称为一个运算符类（简称 opclass）。例如，`int4_ops` 运算符类包含适用于 `int4` 类型的 `= < > > <` 运算符。PostgreSQL 提供了一张叫作 `pg_class` 的系统表，从中可以查到完整的运算符类列表，其中既包含了系统原生支持的类，也包含了通过扩展包机制添加的

类。一种类型的索引会使用特定的若干种运算符类。完整的运算符列表可以从 pgAdmin 界面上的运算符类目下看到，也可以根据 system catalog 在示例 6-7 中执行查询得到。

示例 6-7：查询 B- 树索引支持的数据类型以及运算符类

```
SELECT am.amname AS index_method, opc.opcname AS opclass_name,
opc.opcintype::regtype AS indexed_type, opc.opcdefault AS is_default
FROM pg_am am INNER JOIN pg_opclass opc ON opc.opcmethod = am.oid
WHERE am.amname = 'btree'
ORDER BY index_method, indexed_type, opclass_name;

index_method |    opclass_name     |        indexed_type         | is_default
-------------+---------------------+-----------------------------+------------
btree        | bool_ops            | boolean                     | t
:
btree        | text_ops            | text                        | t
btree        | text_pattern_ops    | text                        | f
btree        | varchar_ops         | text                        | f
btree        | varchar_pattern_ops | text                        | f
 :
```

在示例 6-7 中，仅查询了 B- 树的相关数据。请注意，每类索引都会有多个运算符类，而其中仅有一个会被标记为默认运算符类。如果建立索引时未指定使用哪个运算符类，那么 PostgreSQL 默认会使用默认运算符类。绝大多数情况下这么做是没什么问题的，但并非绝对如此。

例如，B- 树索引默认的 text_ops 运算符类（又名 varchar_ops）中并不支持 ~~ 运算符（即 LIKE 运算符），所以如果建 B- 树索引时选择了该运算符类，那么所有使用 LIKE 的查询都无法在 text_ops 运算符类中使用索引。因此，如果你的业务场景需要对 varchar 或者 text 类型进行大量 LIKE 模糊查询，那么建索引时最好是显式指定使用 text_pattern_ops 或者 varchar_pattern_ops 这两个运算符类。指定运算符类的语法很简单，只需要在建索引时加在被索引字段名的后面即可，参考示例如下：

```
CREATE INDEX idx1 ON census.lu_tracts USING btree (tract_name text_pattern_ops);
```



在上述示例 6-7 的查询结果中，你可能已经注意到了 B- 树索引的 varchar_ops 和 text_ops 两种运算符类适用的数据类型都是 text，这样来看的话，varchar_ops 貌似有点名不副实。这是因为 varchar 类型本质上就是加了长度限制的 text 类型，二者可以共用一套运算符。varchar_ops 和 varchar_pattern_ops 实质上就是 text_ops 和 text_pattern_ops 的别名，之所以把前面两种单列出来是因为 varchar 毕竟是单独的一种数据类型，存在一种以其类型命名的专属运算符类看起来会比较合理。

最后请牢记这一条：你创建的每一个索引都只会使用一个运算符类。如果希望一个字段上的索引使用多个运算符类，那么请创建多个索引。要将默认索引 text_ops 添加到表中，请

运行以下代码：

```
CREATE INDEX idx2 ON census.lu_tracts USING btree (tract_name);
```

现在，在同一个字段上就有了多个索引（单个字段上可建立索引的个数是没有限制的）。规划器处理等值查询时会使用 idx2，处理 like 模糊查询时会使用 idx1。

你可以在 PostgreSQL 官方手册中找到关于运算符类的更详细的描述（http://www.postgresql.org/docs/current/interactive/indexes-opclass.html）。另外我们也强烈建议你阅读一下我们的博客文章“为什么我的索引没发挥作用？”（http://www.postgresonline.com/journal/archives/78-Why-is-my-index-not-being-used.html）。

6.3.3 函数索引

PostgreSQL 的函数索引功能可以基于字段值的函数运算结果建立索引。函数索引的用途也是很广泛的，例如可用于对大小写混杂的文本数据建立索引。PostgreSQL 是一个区分大小写的数据库，如果要实现不区分大小写的查询，那么可以借助如下的函数索引：

```
CREATE INDEX fidx ON featnames_short
 USING btree (upper(fullname) varchar_pattern_ops);
```

建立了该索引之后，类似 SELECT fullname FROM featnames_short WHERE upper(fullname) LIKE 'S%' 这种查询就可以用上索引了。不过要注意，查询语句中使用的函数要与建函数索引时使用的函数完全一致，这样才能保证用上索引。PostgreSQL 和 Oracle 都支持函数索引。MySQL 和 SQL Server 不直接支持函数索引，但提供了自动计算字段并且可以对该类字段建立索引，总的来说其效果和函数索引是类似的。从 9.3 版开始，PostgreSQL 开始支持对物化视图建立索引。

6.3.4 基于部分记录的索引

基于部分记录的索引（有时也称为已筛选索引）是一种仅针对表中部分记录的索引，而且这部分记录需要满足 WHERE 语句设置的筛选条件。例如，假设某表中有 1 000 000 条记录，但你只会查询其中的一个记录数为 10 000 的子集，那么该场景就非常适合使用基于部分记录的索引。这种索引比全量索引要快，因为其体积小，所以可以把更多索引数据缓存到内存中，另外该类索引占用的磁盘空间也会更小。

基于部分记录的索引能够实现仅针对部分记录的唯一性约束。举个例子，假设你手上有一家报纸在过去 10 年间的订阅用户数据，现在需要确保还在订阅的用户们不会每天多拿一份报纸。由于人们对纸媒的兴趣下降，因此这 10 年间的全量订阅用户中仅有 5% 的人还在坚持订阅。所以，很显然你不需要关注那些已经退订的用户，因为他们的姓名早已从报纸递送员手上的递送名单中剔除。表结构如下：

```
CREATE TABLE subscribers (
 id serial PRIMARY KEY,
 name varchar(50) NOT NULL, type varchar(50),
 is_active boolean);
```

我们建立一个基于当前活跃用户的部分记录索引即可：

```
CREATE UNIQUE INDEX uq ON subscribers USING btree(lower(name)) WHERE is_active;
```



索引的 WHERE 条件中使用的函数必须是确定性函数，即固定的输入一定能够得到固定输出的函数。这意味着有几类函数是不能用作筛选条件的：一类是 CURRENT_DATE 这种输出结果不停在变的函数；一类是依赖于其他表数据进行运算的函数，其输出结果受其他表的数据的影响，因此输出也是不固定的；还有一类是依赖当前表中的其他记录行进行运算的函数，其输出也不会受控。

我们需要特别强调的一点是，当使用 SELECT 语句查询数据时，创建索引时所使用的条件必须是你的 WHERE 条件的子集。这看起来比较麻烦也容易出错，那么有一个办法可以让事情变得简单一些，那就是建一个视图，视图条件就是建索引的条件，那么针对此视图进行查询就永远不会漏掉条件了。还是以前述报纸订阅用户数据为例，建立如下视图：

```
CREATE OR REPLACE VIEW vw_subscribers_current AS
SELECT id, lower(name) As name FROM subscribers WHERE is_active = true;
```

然后将针对原表的查询都改为针对此视图的查询（有一种比较激进的观点认为此种情况下永远都不应该直接查询原表）：

```
SELECT * FROM vw_active_subscribers WHERE user_name = 'sandy';
```

你可以查看规划器输出的执行计划来检查你的索引是否被用上了。

6.3.5 多列索引

多列索引也称为复合索引[1]。在本章前面的内容中，你应该已经见到了大量复合索引的例子。另外我们还想介绍的一点是：你可以使用多个基础列创建功能索引。以下是一个多列索引的示例。

```
CREATE INDEX idx ON subscribers USING btree (type, upper(name) varchar_pat
tern_ops);
```

PostgreSQL 的规划器在语句执行过程中会自动使用一种被称为“位图索引扫描”的策略来

注 1：复合索引的多个字段中也可以包含函数，此时建立的索引既是复合索引也是函数索引。
——译者注

同时使用多个索引。该策略可以使得多个单列索引同时发挥作用，达到的效果与使用单个复合索引相同。如果你不能确定业务的应用模式是以单列作为查询条件的场景多一些还是同时以多列作为查询条件的场景多一些，那么最好针对可能作为查询条件的每个列单独建立索引，这样是最灵活的做法，规划器会决定如何组合使用这些索引。

假设你建了一个复合 B- 树索引，其中包含 `type`、`upper(name)` 等多个字段，那么完全没必要针对 `type` 字段再单独建一个索引，因为规划器即使在遇到只有 `type` 单字段的查询条件时也会自动使用该复合索引，这是规划器的一项基本能力。

PostgreSQL 从 9.2 版开始支持仅依赖索引内数据的查询方法，也就是说如果查询的目标字段在索引内都有，那么直接扫描索引就可以得到查询结果，根本不需要访问表的本体了。这个功能的引入使得复合索引的作用更为凸显，因为复合索引可以提供更多数据，因此更适合使用此种查询方法。如果你的业务场景中查询的目标字段和条件字段是相同的那几个，那么就应该建立复合索引以提升查询速度。不过，索引中包含的字段越多也就意味着索引占用的空间会越大，能在内存中缓存的索引条目就越少，因此请不要滥用复合索引。

第7章 PostgreSQL的特色SQL语法

PostgreSQL 在对 ANSI SQL 标准的遵从度方面已经远远走在了其他数据库的前面。除了适配标准之外，PostgreSQL 还提供了类型多样的强化语法，这些强化包括易用的简化语法以及一些前卫到足以打破关系型 SQL 边界的语法特性。通过这些努力，PostgreSQL 保持了持续领先。本章中，我们还将介绍一些其他数据库中很少见的 SQL 语法特性。我们希望你在开始学习本章之前已经具有了一定的 SQL 开发经验，否则你可能无法理解 PostgreSQL 为简化 SQL 开发工作所做的努力。

7.1 视图

关系型数据库中的表存储的是规范化的数据，因此当需要从多张表中取数据时，就需要写关联查询的 SQL 语句。如果你的应用场景需要反复执行这种关联查询语句，可以考虑创建一个视图。简单来说，视图就是持久化存储在数据库中的一个查询语句。

有人认为用户不应该直接访问表，而应该通过视图来访问，不过这就意味着需要为每张表都创建一个视图。这种做法的优点是在表的本体之上增加了一个访问层，方便了权限控制和业务逻辑抽象，缺点就是太麻烦。我们认为这个观点是合理的，但事实上由于惰性很少有人会这么做。

PostgreSQL 的视图功能近年来有了长足的发展。在 9.1 版之前，要想对视图进行更新操作，唯一的方法就是使用规则。“使用可更新视图来对数据库进行抽象”（http://www.postgresonline.com/journal/archives/11-Database-Abstraction-with-updatable-Views.html）这篇博文中有一个关于规则的例子。在 9.1 版及之后的版本中，你依然可以通过创建规则来更

新视图，但我们推荐使用 INSTEAD OF 触发器来实现该功能，这是业界标准做法，其他数据库一般也都这么实现。

9.3 版中推出了可自动更新的视图。如果你的视图是从单个表得出的，并且你将主键作为一个输出列，那么就可以直接对此视图发出 UPDATE 命令。基础表将存储该更新。

9.3 版中还引入了*物化视图*。每个视图都对应一个 SQL 查询语句，视图本质上就是该 SQL 的查询结果集的一个别名。每次访问视图时都需要执行其对应的 SQL，但物化视图将视图逻辑映射后的数据记录实际存储下来，这样访问物化视图时就省略了视图底层 SQL 的执行过程，就像访问一张本地表一样。一旦物化视图建立好以后，只有对它执行 REFRESH 操作时才会再次从基础表中读取数据。根据前面的描述可以知道，使用物化视图可以节省计算资源，因为视图底层 SQL（这种 SQL 逻辑可能极其复杂）不用反复执行。但物化视图也有缺点，因为如果刷新不及时就会导致取出的数据可能不是最新的。

9.4 版开始支持用户在物化视图刷新时也能对其进行访问，该版本还引入了 WITH CHECK OPTION 修饰符，用于防止在视图的范围之外进行插入和更新。

7.1.1　单表视图

最简单的视图是从单个表得出的。如果打算将数据写回到该表，请始终包含主键，如示例 7-1 所示。

示例 7-1：创建基于单表的视图

```
CREATE OR REPLACE VIEW census.vw_facts_2011 AS
SELECT fact_type_id, val, yr, tract_id FROM census.facts WHERE yr = 2011;
```

自从 9.3 版起，就可以使用 INSERT、UPDATE 和 DELETE 命令在该视图中更改数据了。更新和删除命令将遵从作为视图一部分的任何 WHERE 条件。例如，下面的删除命令将仅删除 yr=2011 的记录：

```
DELETE FROM census.vw_facts_2011 WHERE val = 0;
```

以下 UPDATE 操作更新不了任何记录：

```
UPDATE census.vw_facts_2011 SET val = 1 WHERE val = 0 AND yr = 2012;
```

注意你可以插入和更新将该视图置于视图的 WHERE 条件之外的数据：

```
UPDATE census.vw_facts_2011 SET yr = 2012 WHERE yr = 2011;
```

上述 UPDATE 语句操作的目标记录是落在视图可见范围内的，但更新后会把本来落在视图可见范围内的记录变成了落到视图可见范围之外，也就是说视图更新后少了一条记录。但为了保持视图数据的一致性，我们不希望这种情况发生，也就是说希望更新后的数据仍然应

该落在视图可见范围内。这可以通过 9.4 版中引入的 WITH CHECK OPTION 修饰符来实现。创建视图时如果包含此修饰符，那么此视图中插入的数据或者更新后的数据落在视图可见范围之外时，系统会报错，违反了该约束的操作会失败。下面我们将限定 vs_facts_2011 视图仅允许插入 2011 年的数据，同时不允许将 yr 字段修改为 2011 以外的其他值。我们修改一下视图定义把这个约束加上，语法如示例 7-2 所示。

示例 7-2：创建带有 WITH CHECK OPTION 约束的单表视图

```
CREATE OR REPLACE VIEW census.vw_facts_2011 AS
SELECT fact_type_id, val, yr, tract_id
FROM census.facts WHERE yr = 2011 WITH CHECK OPTION;
```

尝试执行以下更新操作：

```
UPDATE census.vw_facts_2011 SET yr = 2012 WHERE val > 2942;
```

你会看到这样的报错信息：

```
ERROR:  new row violates WITH CHECK OPTION for view"vw_facts_2011"
DETAIL:  Failing row contains (1, 25001010500, 2012, 2985.000, 100.00).
```

7.1.2 使用触发器来更新视图

视图可以将针对多张表的关联查询封装为针对视图的简单查询。如果视图的基表有多张，那么直接更新该视图是不允许的，因为多张表必然带来的问题就是操作要落到哪个基表上，PostgreSQL 是无法自动判定的。假设你有一个视图，该视图基于一张国家信息表和一张省份信息表，此时你希望删除该视图的一条记录，PostgreSQL 无法得知你到底想要仅删除一条国家记录，还是仅删除一个省份记录，还是删除一个国家以及该国家对应的所有省的记录。PostgreSQL 无法自动判定你想做什么并不代表就不能对这种复杂视图进行修改操作，你可以通过编写触发器来对这些操作进行转义处理，转义后的逻辑中可以体现你的意图。

我们首先建立一个关联了两张表的视图，如示例 7-3 所示。

示例 7-3：创建 vw_facts 视图

```
CREATE OR REPLACE VIEW census.vw_facts AS
SELECT y.fact_type_id, y.category, y.fact_subcats, y.short_name, x.tract_id, x.yr,
x.val, x.perc
FROM census.facts As x INNER JOIN census.lu_fact_types As y
ON x.fact_type_id = y.fact_type_id;
```

然后可以定义一个或者多个 INSTEAD OF 触发器来实现针对 INSERT、UPDATE、DELETE 这三大基本操作的转义处理。触发器需要有一个基础函数，你可以使用任何语言来编写该基础函数，其命名也没有规则限制。我们在示例 7-4 中选择使用 PL/pgSQL 语法来编写。

示例 7-4：在 vw_facts 视图上建一个对 insert、update、delete 操作进行转义处理的函数

```
CREATE OR REPLACE FUNCTION census.trig_vw_facts_ins_upd_del() RETURNS trigger AS
$$
BEGIN
    IF (TG_OP = 'DELETE') THEN ❶
        DELETE FROM census.facts AS f
        WHERE
            f.tract_id = OLD.tract_id AND f.yr = OLD.yr AND
            f.fact_type_id = OLD.fact_type_id;
        RETURN  OLD;
    END IF;
    IF (TG_OP = 'INSERT') THEN ❷
        INSERT INTO census.facts(tract_id, yr, fact_type_id, val, perc)
        SELECT NEW.tract_id, NEW.yr, NEW.fact_type_id, NEW.val, NEW.perc;
        RETURN NEW;
    END IF;
    IF (TG_OP = 'UPDATE') THEN ❸
        IF
            ROW(OLD.fact_type_id, OLD.tract_id, OLD.yr, OLD.val, OLD.perc) !=
            ROW(NEW.fact_type_id, NEW.tract_id, NEW.yr, NEW.val, NEW.perc)
        THEN ❹
            UPDATE census.facts AS f
            SET
                tract_id = NEW.tract_id,
                yr = NEW.yr,
                fact_type_id = NEW.fact_type_id,
                val = NEW.val,
                perc = NEW.perc
            WHERE
                f.tract_id = OLD.tract_id AND
                f.yr = OLD.yr AND
                f.fact_type_id = OLD.fact_type_id;
            RETURN NEW;
        ELSE
            RETURN NULL;
        END IF;
    END IF;
END;
$$
LANGUAGE plpgsql VOLATILE;
```

❶ 对删除操作进行转义处理，筛选条件的字段取值来源于 OLD 记录。[1]

❷ 对插入操作进行转义处理。

❸ 对更新操作进行转义处理。根据 OLD 记录的内容判断哪些记录要更新为 NEW 记录。[2]

❹ 比较 OLD 记录和 NEW 记录的字段值，只有二者不一样时才真正执行更新动作。

注 1：OLD 记录是指原始的针对视图的删除动作所要删除的视图记录。也就是说，OLD 记录是视图记录，而非视图基础表的记录。——译者注

注 2：NEW 记录指的是原始的针对视图的更新动作设置的修改后的视图记录。也就是说，NEW 记录是视图记录，而非视图基础表的记录。——译者注

接下来，我们将此触发器函数绑定到视图上，语法如示例 7-5 所示。

示例 7-5：将触发器函数绑定到视图上

```
CREATE TRIGGER census.trig_01_vw_facts_ins_upd_del
INSTEAD OF INSERT OR UPDATE OR DELETE ON census.vw_facts
FOR EACH  ROW EXECUTE PROCEDURE census.trig_vw_facts_ins_upd_del();
```

现在针对视图进行更新、删除或插入操作时，这些操作将更新基础 facts 表：

```
UPDATE census.vw_facts SET yr = 2012 WHERE yr = 2011 AND tract_id =
'25027761200';
```

执行后会输出一条注释：

```
Query returned successfully: 56 rows affected, 40 ms execution time.
```

如果我们试图更新的字段不在更新行比较列表中（如此处所示），那么更新操作就不会命中任何记录，因为入口条件不满足：

```
UPDATE census.vw_facts SET short_name = 'test';
```

输出消息将如下所示：

```
Query returned successfully: 0 rows affected, 931 ms execution time.
```

前面的例子中我们用一个触发器函数处理了所有类型的触发事件（insert、update、delete），但实际上专门为每种触发事件创建一个独立的触发器也是可以的。

7.1.3　物化视图

物化视图会把视图可见范围内的数据在本地缓存下来，然后就可以当成一张本地表来使用。首次创建物化视图以及对其执行 REFRESH MATERIALIZED VIEW 刷新操作时都会触发数据缓存动作，只不过前者是全量缓存，后者是增量刷新。请注意物化视图特性是从 9.3 版开始支持的。

物化视图最典型的应用场景是用于加速时效性要求不高的长时复杂查询，在 OLAP（在线分析与处理）领域，这种查询是经常出现的。

物化视图还有一个特点就是支持建立索引以加快查询速度。示例 7-6 建立了示例 7-1 中的视图的物化版本。

示例 7-6：建立物化视图

```
CREATE MATERIALIZED VIEW census.vw_facts_2011_materialized AS
SELECT fact_type_id, val, yr, tract_id FROM census.facts WHERE yr = 2011;
```

然后对物化视图建立一个索引，语法与在普通表上建索引完全相同，如示例 7-7 所示。

示例 7-7：在物化视图上建立索引

```
CREATE UNIQUE INDEX ix
ON census.vw_facts_2011_materialized (tract_id, fact_type_id, yr);
```

当物化视图中含大量记录时，为了加快对它的访问速度，我们需要对数据进行排序。要实现这一点，最简单的方法就是在创建物化视图时使用的 SELECT 语句中增加 ORDER BY 子句。另外一种方法就是对其执行聚簇排序操作以使得记录的物理存储顺序与索引的顺序相同，具体步骤是：首先创建一个索引，该索引应体现你所希望的排序；然后基于指定索引对物化视图执行 CLUSTER 命令，语法如示例 7-8 所示。

示例 7-8：基于某索引对物化视图执行聚簇排序操作

```
CLUSTER census.vw_facts_2011_materialized USING ix; ❶
CLUSTER census.vw_facts_2011_materialized; ❷
```

❶ 指定聚簇操作所依据的索引名。执行过以后系统就会自动记下该表是依据哪个索引进行聚簇排序的，后面再次执行聚簇操作时系统会自动使用该索引，所以索引名仅在首次聚簇操作时需要，后续不再需要。

❷ 每次刷新过物化视图后，都需要重新对其进行一次聚簇排序操作。

相对于 CLUSTER 方案来说，ORDER BY 方案的优点在于每次执行 REFRESH MATERIALIZED VIEW 时都会自动对记录进行重排序，但 CLSUTER 方案就必须手动执行；其缺点在于物化视图加了 ORDRE BY 以后，REFRESH 操作执行会耗时更久。在正式使用 ORDER BY 方案之前，你应该对 REFRESH 操作进行测试，看其性能是否可接受。另一种测试方法是直接运行创建物化视图时所用的带 ORDER BY 的 SQL 语句。

在 PostgreSQL 9.3 版中，刷新物化视图的语法如下：

```
REFRESH MATERIALIZED VIEW census.vw_facts_2011_materialized;
```

在 PostgreSQL 9.4 版中，为了解决物化视图刷新操作导致的锁表问题，可以使用以下语法：

```
REFRESH MATERIALIZED VIEW CONCURRENTLY census.vw_facts_2011_materialized;
```

物化视图有以下几个缺点。

- 不支持通过 CREATE OR REPLACE 语法来对一个现有的物化视图进行重建，要想重建只能先删除再重建，即使只做很小的改变也需要这么干。删除语法是 DROP MATERIALIZED VIEW+ 视图名。删除以后，该视图上所有的索引都会丢失。
- 每次刷新数据时都要执行一次 REFRESH MATERIALIZED VIEW 操作。PostgreSQL 不支持自动刷新物化视图。要想实现自动刷新，你必须使用类似 crontab、pgAgent 定时任务或者是触发器之类的机制。我们在“使用物化视图和语句级触发器来实现数据缓存”这篇博文（http://www.postgresonline.com/journal/archives/313-Caching-data-with-materialized-views-and-statement-level-triggers.html）中提供了一个使用触发器来进行刷新的例子，可

供你参考。

- 在 9.3 版中，刷新物化视图是一个阻塞操作；也就是说，视图在刷新期间是无法访问的。9.4 版中引入了一个新的 CONCURRENTLY 关键字，在 REFRESH 命令中增加了该关键字以后，可以解除锁定限制，前提条件是被刷新的物化视图上要有一个唯一索引。实现无锁刷新的代价就是如果有人在刷新期间进行视图访问，那么刷新时间会变长。

7.2 灵活易用的PostgreSQL专有SQL语法

我们在多年编写 SQL 语句的过程中使用过很多 PostgreSQL 的专有语法，借助它们可以编写出更加简洁以及功能更加强大的 SQL。我们在本节中介绍的这些语法都是 PostgreSQL 的专有语法。“专有”意味着该语法不符合 ANSI SQL 标准。如果你的老板要求编写的 SQL 必须遵守 ANSI SQL 标准，又或者你的 SQL 需要移植到别的数据库上去，那么请不要使用本节介绍的这些专有语法。

7.2.1 DISTINCT ON

我们认为最好用的一个专有语法就是 DISTINCT ON，其功能类似 DISTINCT 但却可以精确到更细的粒度。DISTINCT 会将结果集中完全重复的记录剔除，但 DISTINCT ON 可以将结果集中指定字段值的重复记录剔除，具体实现方法是先对结果集按照 DISTINCT ON 指定的字段进行排序，然后筛选出每个字段值第一次出现时所在的记录，其余的记录都剔除。可以看到一个小小的单词 ON 就实现了必须写大量代码才能实现的功能。

在示例 7-9 中，我们演示了如何获取马萨诸塞州每个县的第一个人口统计区的信息。

示例 7-9：DISTINCT ON 的用法

```
SELECT DISTINCT ON (left(tract_id, 5))
    left(tract_id, 5) As county, tract_id, tract_name
FROM census.lu_tracts
ORDER BY county, tract_id;

county |  tract_id   |                    tract_name
-------+-------------+---------------------------------------------------
25001  | 25001010100 | Census Tract 101, Barnstable County, Massachusetts
25003  | 25003900100 | Census Tract 9001, Berkshire County, Massachusetts
25005  | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007  | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009  | 25009201100 | Census Tract 2011, Essex County, Massachusetts
:
```

请注意，ON 修饰符支持设置多列，运算时将基于这多个列的总体唯一性来进行去重操作。同时查询语句中 ORDER BY 子句的排序字段列表的最左侧必须是 DISTINCT ON 指定的字段列表，即保证整个结果集是按照这几个字段排序的，这样最终去重后得到的结果才是你想要的。

7.2.2 LIMIT和OFFSET关键字

LIMIT 关键字指定了查询时仅返回指定数量的记录，OFFSET 关键字指定了从第几条记录开始返回。你可以将二者结合起来使用也可以单独使用。一般来说，这两个关键字总是和 ORDER BY 联用的，因为只有在一个已经按照用户的意图排好序的结果集上指定返回特定的子结果集才有意义。在示例 7-10 中，我们演示了 OFFSET 关键字的用法，如果不设置 OFFSET 的话，其值默认为 0。

该语法并非 PostgreSQL 所特有，事实上它最早源自于 MySQL。这种限制返回结果记录数的功能在很多数据库中都支持，但具体语法和内部实现机制各有千秋。

示例 7-10：要求示例 7-9 的查询结果集仅返回从第 3 条开始的 3 条记录

```
SELECT DISTINCT ON (left(tract_id, 5))
    left(tract_id, 5) As county, tract_id, tract_name
FROM census.lu_tracts
ORDER BY county, tract_id LIMIT 3 OFFSET 2;

county |  tract_id   |                    tract_name
-------+-------------+---------------------------------------------------
25005  | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007  | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009  | 25009201100 | Census Tract 2011, Essex County, Massachusetts
```

7.2.3 简化的类型转换语法

ANSI SQL 标准中定义了一个名为 CAST 的类型转换函数，可以实现数据类型之间的互转。例如 CAST('2011-1-11' AS date) 可以将文本 2011-1-1 转换为一个日期型数据。PostgreSQL 支持一种简写语法，该语法使用了两个冒号来表示转换关系，具体格式为：'2011-1-1'::date。这种写法形式更简洁也更易于使用，比如有时候我们需要级联执行多个类型转换动作，也就是说需要将类型 A 转换为类型 B 再转为类型 C，这种情况用简写语法也是可以实现的，例如 someXML::text::integer。

7.2.4 一次性插入多条记录

PostgreSQL 支持一次性插入多条记录的语法。示例 7-11 演示了如何向我们在示例 6-2 中创建的表中一次性插入多条记录。

示例 7-11：一次性插入多条记录

```
INSERT INTO logs_2011 (user_name, description, log_ts)
VALUES
    ('robe', 'logged in', '2011-01-10 10:15 AM EST'),
    ('lhsu', 'logged out', '2011-01-11 10:20 AM EST');
```

请注意，在 PostgreSQL 中 VALUES 子句并不是只能作为 INSERT 语句的一部分来使用，它其实是一个动态生成的临时结果集，可用于多种场合，如示例 7-12 所示。

示例 7-12：使用 VALUES 语法来模拟一个虚拟表

```
SELECT *
FROM (
    VALUES
        ('robe', 'logged in', '2011-01-10 10:15 AM EST'::timestamptz),
        ('lhsu', 'logged out', '2011-01-11 10:20 AM EST'::timestamptz)
    ) AS l (user_name, description, log_ts);
```

将 VALUES 子句当作一个虚拟表来用时，需要为该表指定字段名，并将那些无法隐式转换的字段值显式地进行类型转换。

7.2.5 使用ILIKE实现不区分大小写的查询

PostgreSQL 是一套区分大小写的系统，如果要实现不区分大小写的文本搜索，有两种方法：一种是将 ANSI LIKE 运算符两边的文本都用 upper 函数转为大写，但这样会导致用不上索引，或者必须单独建立一个基于 upper 函数的函数式索引才能使查询语句用上索引；另一种是使用 PostgreSQL 所特有的 ILIKE 运算符（~），语法如下：

```
SELECT tract_name FROM census.lu_tracts WHERE tract_name ILIKE '%duke%';

tract_name
------------------------------------------------
Census Tract 2001, Dukes County, Massachusetts
Census Tract 2002, Dukes County, Massachusetts
Census Tract 2003, Dukes County, Massachusetts
Census Tract 2004, Dukes County, Massachusetts
Census Tract 9900, Dukes County, Massachusetts
```

7.2.6 可以返回结果集的函数

PostgreSQL 允许返回集的函数显示在 SQL 语句的 SELECT 子句中。许多其他数据库都不支持该特性，在这些数据库中仅会在 SELECT 子句中显示标量函数。

在一个复杂的 SQL 语句中使用返回结果集的函数很容易就会导致意外的结果，这是因为这类函数输出的结果集会与该语句其他部分生成的结果集产生笛卡儿积，从而生成更多的记录行。因此在使用之前你必须对此后果有所了解。在示例 7-13 中，我们使用 generate_series 函数演示了这种情况。先建表如下：

```
CREATE TABLE interval_periods (i_type interval);
INSERT INTO interval_periods (i_type)
VALUES('5 months'), ('132 days'), ('4862 hours');
```

示例 7-13：在 SELECT 语句中使用返回结果集的函数

```
SELECT i_type,
    generate_series('2012-01-01'::date,'2012-12-31'::date,i_type) As dt
FROM interval_periods;
```

```
i_type     |           dt
-----------+------------------------
5 months   | 2012-01-01 00:00:00-05
5 months   | 2012-06-01 00:00:00-04
5 months   | 2012-11-01 00:00:00-04
132 days   | 2012-01-01 00:00:00-05
132 days   | 2012-05-12 00:00:00-04
132 days   | 2012-09-21 00:00:00-04
4862 hours | 2012-01-01 00:00:00-05
4862 hours | 2012-07-21 15:00:00-04
```

7.2.7 限制对继承表的DELETE、UPDATE、INSERT操作的影响范围

如果表间是继承关系，那么查询父表时就会将子表中满足条件的记录也查出来。DELETE 和 UPDATE 操作也遵循类似逻辑，即对父表的修改操作也会影响子表的记录。有时你可能希望操作仅限定于主表范围之内而并不希望子表受到波及。

PostgreSQL 提供了 ONLY 关键字以实现此功能。我们在示例 7-30 中展示了 ONLY 的用法。在该示例中，我们希望仅从生产表中删除那些尚未迁移到日志表中的记录。如果没有 ONLY 修饰符，我们最终将从子表中删除先前可能已移动过的记录。

7.2.8 DELETE USING语法

我们经常会遇到“只有当记录的字段值落在另外一个结果集中时才需要删除该记录”的情况，那么此时就必须借助一次关联查询才能定位到要删除的目标记录。USING 子句可以将需要借助的一个或者多个中间表（或者子查询）纳入同一个 DELETE 语句中。在示例 7-14 中，我们借助一个关联查询实现了删除 census.facts 表中符合 short_name='s01' 这个条件的记录。

示例 7-14：DELETE USING 的用法

```
DELETE FROM census.facts
USING census.lu_fact_types As ft
WHERE facts.fact_type_id = ft.fact_type_id AND ft.short_name = 's01';
```

而符合标准的方式将会是，在 WHERE 子句中使用笨拙的 IN 表达式。

7.2.9 将修改影响到的记录行返回给用户

RETURNING 是 ANSI SQL 规定的标准语法，但支持该语法的数据库却不多。在示例 7-30 中，我们通过 RETURNING 子句将在 DELETE 操作中被删除的记录返回给了用户。当然，INSERT 和 UPDATE 操作也是可以使用 RETURNING 的。对于带 serial 类型字段的表来说，RETURNING 语法是很有用的，因为向这类表中插入记录时，serial 字段是临时生成而非用户指定的。也就是说在插入动作完成之前，用户也不知道 serial 字段的值会是多少，除非是再查询一

遍。而 RETURNING 语法使得用户不用再次查询就立即得到了 serial 字段的值。最常见的用法一般是 RETURNING *，即返回所有字段的值，但也可以指定仅返回特定字段，如示例 7-15 所示。

示例 7-15：在 UPDATE 语句中使用 RETURNING 子句返回修改过的记录

```
UPDATE census.lu_fact_types AS f
SET short_name = replace(replace(lower(f.fact_subcats[4]),' ','_'),':','')
WHERE f.fact_subcats[3] = 'Hispanic or Latino:' AND f.fact_subcats[4] > ''
RETURNING fact_type_id, short_name;

fact_type_id  |                  short_name
--------------+-------------------------------------------------
96            | white_alone
97            | black_or_african_american_alone
98            | american_indian_and_alaska_native_alone
99            | asian_alone
100           | native_hawaiian_and_other_pacific_islander_alone
101           | some_other_race_alone
102           | two_or_more_races
```

7.2.10　在查询中使用复合数据类型

PostgreSQL 会在建表时自动创建一个结构与表完全相同的数据类型，其中包含了多个其他数据类型的成员字段，因此也会被称为复合数据类型。你第一次见到基于复合数据类型的查询语句时可能会感到很惊讶。事实上，你可能已经在编写调试 SQL 语句的过程中见识过它的神奇之处。先看一下这个语句：

```
SELECT x FROM census.lu_fact_types As x LIMIT 2;
```

第一眼看到这个语句时，你可能认为我们漏写了一个“.*”，但请看一下该语句的执行结果：

```
x
------------------------------------------------------------------
(86,Population,"{D001,Total:}",d001)
(87,Population,"{D002,Total:,""Not Hispanic or Latino:""}",d002)
```

语句不但没有报错，而且返回的结果是标准的 lu_fact_type 类型。我们来看一下第一行记录的内容来确认有没有问题，86 是 fact_type_id 字段的值，Population 是 category 字段的值，{D001,Total:} 是 fact_subcats 属性。可以看到，字段值与表定义完全匹配，没有问题。复合数据类型可以作为多个很有用的函数的输入，比如 array_agg 和 hstore（hstore 扩展包提供的一个函数，可以将一行记录转换为 hstore 的一个键值对象）等。

如果你的项目正在使用 Ajax 技术和 PostgreSQL 数据库，并且使用的是 PostgreSQL 9.2 版或者更新的版本，那么就可以充分利用 PostgreSQL 对 JSON 类型的原生支持能力，通过联用 array_agg 和 array_to_json 这两个函数来将语句的查询结果转换为一个 JSON 对象后输

出，如示例 7-16 所示。

示例 7-16：将查询结果转换为 JSON 格式

```
SELECT array_to_json(array_agg(f)) As cat ❶
FROM (
    SELECT MAX(fact_type_id) As max_type, category ❷
    FROM census.lu_fact_types
    GROUP BY category
) As f;
```

输出结果如下：

```
cats
----------------------------------------------------
[{"max_type":102,"category":"Population"},
{"max_type":153,"category":"Housing"}]
```

❶ 将子查询 f 中的所有记录转换为一个基于复合数据类型的数组。

❷ 定义一个名为 f 的子查询，可以从中查出记录。

PostgreSQL 9.3 版提供了一个名为 json_agg 的函数，该函数的效果相当于上面示例中 array_to_json 和 array_agg 联用的效果，但 json_agg 执行速度更快。在示例 7-17 中，我们使用 json_agg 改写了示例 7-16 中的语句，二者的输出是相同的。

示例 7-17：使用 json_agg 将查询结果转为 JSON 格式

```
SELECT json_agg(f) As cats
FROM (
    SELECT MAX(fact_type_id) As max_type, category
    FROM census.lu_fact_types
    GROUP BY category
) As f;
```

7.2.11　DO

DO 命令可以执行一个基于过程化语言的匿名代码段。在下面的示例中，我们将演示如何通过执行一个匿名代码段来将示例 3-7 中插入的数据从中间表加载到产品表中。例子中的匿名代码段是用 PL/pgSQL 编写的，但你也可以使用别的语言编写。

示例 7-18 中生成了一系列的 INSERT INTO SELECT 语句，然后通过这些语句实现数据迁移，这些 SQL 还实现了由列转行的转换操作。



示例 7-18 中的 lu_fact_types 建表语句中仅包含了部分字段。请从本书附加的代码和数据资源包中找到 building_census_tables.sql 这个脚本，其中有完整的建表语句。

示例 7-18：使用 DO 命令来生成动态 SQL

```
set search_path=census;
DROP TABLE IF EXISTS lu_fact_types;
CREATE TABLE lu_fact_types (
    fact_type_id serial,
    category varchar(100),
    fact_subcats varchar(255)[],
    short_name varchar(50),
    CONSTRAINT pk_lu_fact_types PRIMARY KEY(fact_type_id)
);

DO language plpgsql
$$
DECLARE var_sql text;
BEGIN
    var_sql := string_agg(
        'INSERT INTO lu_fact_types(category, fact_subcats, short_name)
        SELECT
            ''Housing'',
            array_agg(s' || lpad(i::text,2,'0') || ') As fact_subcats,
            ' || quote_literal('s' || lpad(i::text,2,'0')) || ' As short_name
        FROM staging.factfinder_import
        WHERE s' || lpad(I::text,2,'0') || ' ~ ''^[a-zA-Z]+'' ', ';'
    )
    FROM generate_series(1,51) As I; ❶
    EXECUTE var_sql; ❷
END
$$;
```

❶ 使用 string_agg 函数让一组 SQL 语句形成单一字符串的形式 INSERT INTO lu_fact_type(...) SELECT ... WHERE s01 ~ '[a-zA-Z]+';。

❷ 执行该 SQL。

7.3 适用于聚合操作的FILTER子句

9.4 版中新引入了用于聚合操作的 FILTER 子句，这是近期 ANSI SQL 标准中新加入的一个关键字。该关键字用于替代同为 ANSI SQL 标准语法的 CASE WHEN 子句，使聚合操作的语法得以简化。例如，假设你需要使用 CASE WHEN 子句来统计每个学生不同科目的多次测试的平均成绩，语法如下。

示例 7-19：在 AVG 聚合函数中使用 CASE WHEN

```
SELECT student,
    AVG(CASE WHEN subject ='algebra' THEN score ELSE NULL END) As algebra,
    AVG(CASE WHEN subject ='physics' THEN score ELSE NULL END) As physics
FROM test_scores
GROUP BY student;
```

用 FILTER 子句可以实现与上面语句等价的效果，语法如示例 7-20 所示。

示例 7-20：AVG 聚合函数与 FILTER 子句的配合使用

```
SELECT student,
    AVG(score) FILTER (WHERE subject ='algebra') As algebra,
    AVG(score) FILTER (WHERE subject ='physics') As physics
FROM test_scores
GROUP BY student;
```

对于求平均值、求合计值以及其他很多聚合函数来说，CASE 和 FILTER 子句是等价的，即二者可以起到相同的作用。FILTER 子句的优势在于写法比较清晰简洁并且操作大数据量时速度比较快。CASE 语句对于筛选掉的字段值是当成 NULL 处理的，因此对于 array_agg 这种会处理 NULL 值的聚合函数来说，使用 CASE WHEN 子句就不止是写法繁琐的问题了，还会导致输出不想要的结果。在示例 7-21 中，我们使用 CASE...WHEN... 方法查询每个学生的各门课程的多次测试成绩的列表来向你演示这个问题。

示例 7-21：CASE WHEN 子句与 array_agg 函数配合使用

```
SELECT student,
    array_agg(CASE WHEN subject ='algebra' THEN score ELSE NULL END) As algebra,
    array_agg(CASE WHEN subject ='physics' THEN score ELSE NULL END) As physics
FROM test_scores
GROUP BY student;

 student |          algebra          |      physics
---------+---------------------------+-------------------------------
 jojo    | {74,NULL,NULL,NULL,74,..} | {NULL,83,NULL,NULL,NULL,79,..}
 jdoe    | {75,NULL,NULL,NULL,78,..} | {NULL,72,NULL,NULL,NULL,72..}
 robe    | {68,NULL,NULL,NULL,77,..} | {NULL,83,NULL,NULL,NULL,85,..}
 lhsu    | {84,NULL,NULL,NULL,80,..} | {NULL,72,NULL,NULL,NULL,72,..}
(4 rows)
```

可以看到上面输出的成绩列表中含有很多的 NULL 值。这个问题可以通过使用子查询来解决，但比起使用 FILTER 来说还是麻烦又低效。示例 7-22 中演示了使用 FILTER 时的写法。

示例 7-22：FILTER 子句与 array_agg 函数的配合使用

```
SELECT student,
    array_agg(score) FILTER (WHERE subject ='algebra') As algebra,
    array_agg(score) FILTER (WHERE subject ='physics') As physics
FROM test_scores
GROUP BY student;

 student | algebra | physics
---------+---------+--------
 jojo    | {74,74} | {83,79}
 jdoe    | {75,78} | {72,72}
 robe    | {68,77} | {83,85}
 lhsu    | {84,80} | {72,72}
```

FILTER 子句适用于所有聚合函数，不仅仅是 PostgreSQL 中内置的那些聚合函数，通过安装扩展包支持的聚合函数也是可以用的。

7.4 窗口函数

PostgreSQL 从 8.4 版开始支持 ANSI SQL 标准中规定的窗口函数特性。通过使用窗口函数，可以在当前记录行中访问到与其存在特定关系的其他记录行，相当于在每行记录上都开了一个访问外部数据的窗口，这也是“窗口函数”这个名称的由来。“窗口”就是当前行可见的外部记录行的范围。通过窗口函数可以把当前行的“窗口”区域内的记录的聚合运算结果附加到当前记录行。`row_number` 和 `rank` 这类窗口函数能够基于窗口区的数据实现对记录行的复杂排序。

如果不借助窗口函数而又想要达到相同的效果，就只能使用关联操作和子查询来实现。表面上看，使用窗口函数违背了 SQL 语言“基于结果集”的编程思想，因为它为每一行数据拓展出了一个外部数据域。但从另外一个角度看，我们可以认为窗口函数本质上仅是一种用来替代关联操作和子查询的简写语法，也就是说窗口函数并未突破 SQL 体系原有的运算逻辑，那么这样就不算是违反了“基于结果集”的思想。你可以从 PostgreSQL 官方手册的“窗口函数”这一节（http://www.postgresql.org/docs/current/interactive/tutorial-window.html）中看到更多说明和示例。

示例 7-23 中通过一个简单的例子来帮助你理解窗口函数的基本概念。通过使用窗口函数，我们可以在单个 `SELECT` 语句中同时获取到符合 `fact_type_id=86` 条件的记录的均值计算结果以及原始记录的详细信息。请注意，语句执行时总是先筛选 `WHERE` 条件再计算窗口函数的，因为这样显然可以避免做无用功。

示例 7-23：基本的窗口函数

```
SELECT tract_id, val, AVG(val) OVER () as val_avg
FROM census.facts
WHERE fact_type_id = 86;

 tract_id   |    val    |        val_avg
------------+-----------+----------------------
25001010100 |  2942.000 | 4430.0602165087956698
25001010206 |  2750.000 | 4430.0602165087956698
25001010208 |  2003.000 | 4430.0602165087956698
25001010304 |  2421.000 | 4430.0602165087956698
:
```

`OVER` 子句限定了窗口中的可见记录范围。本例中的 `OVER` 子句未设定任何条件，因此从该窗口中能看见全表所有记录，所以 `AVERAGE` 运算的结果就是表中所有符合 `fact_type_id=86` 条件的记录中 `val` 字段的平均值。你可以看到，通过为其增加 `OVER` 子句，我们把一个传统的 AVG 聚合运算函数转变成了一个窗口函数。PostgreSQL 在遍历每一行记录时都会基于全表记录进行一次 AVG 运算，然后将得到的均值作为当前行的一个字段输出。由于窗口数据域内包含多条记录，这意味着窗口函数运算的结果一定会在多条记录上都是重复的。事实上，窗口函数实现了无需 `GROUP BY` 的聚合运算，还实现了无需 `JOIN` 的关联操作，从

而将窗口函数的运算结果回填到记录行中。

所有 SQL 聚合函数都可以通过增加 OVER 子句的方式来当作窗口函数使用。除了这些双重身份的函数之外，系统中还有 ROW、RANK、LEAD 等专门的窗口函数，你可以从 PostgreSQL 官方手册的“窗口函数”一节（http://www.postgresql.org/docs/current/interactive/functions-window.html）中看到完整的窗口函数列表。

7.4.1 PARTITION BY子句

窗口函数的窗口可见记录范围是可设置的，可以是全表记录，也可以是与当前行有关联关系的特定记录行。窗口可见记录范围的设置是通过 PARTITION BY 子句实现的，它可以指示 PostgreSQL 仅在满足条件的特定记录集上执行聚合操作。示例 7-24 的查询与示例 7-23 类似，但要求各县级编号作为窗口筛选条件，该编号就是 tract_id 的前 5 个字符。

示例 7-24：使用县级编号作为窗口可见记录范围的筛选条件

```
SELECT tract_id, val, AVG(val) OVER (PARTITION BY left(tract_id,5)) As val_avg_coun
ty
FROM census.facts WHERE fact_type_id = 2 ORDER BY tract_id;

  tract_id   |   val    |    val_avg_county
-------------+----------+-------------------
 25001010100 | 1765.000 | 1709.9107142857142857
 25001010206 | 1366.000 | 1709.9107142857142857
 25001010208 |  984.000 | 1709.9107142857142857
 :
 25003900100 | 1920.000 | 1438.2307692307692308
 25003900200 | 1968.000 | 1438.2307692307692308
 25003900300 | 1211.000 | 1438.2307692307692308
 :
```

（请注意：为了节约版面，上面仅截取了完整输出的部分内容。）

7.4.2 ORDER BY子句

窗口函数的 OVER 子句中还可以使用 ORDER BY 子句，其作用可以理解为对窗口可见范围内的所有记录进行排序，并且窗口可见记录域是从结果集的第一条记录开始到当前记录为止的范围内。该语法的典型应用场景就是用 ROW_NUMBER 函数对记录集进行编号。在示例 7-25 中，我们演示了如何对各人口普查区记录按照其名称顺序进行编号。

示例 7-25：使用 ROW_NUMBER 窗口函数进行编号操作

```
SELECT ROW_NUMBER() OVER (ORDER BY tract_name) As rnum, tract_name
FROM census.lu_tracts
ORDER BY rnum LIMIT 4;

rnum |                    tract_name
-----+-------------------------------------------------
```

```
1     | Census Tract 1, Suffolk County, Massachusetts
2     | Census Tract 1001, Suffolk County, Massachusetts
3     | Census Tract 1002, Suffolk County, Massachusetts
4     | Census Tract 1003, Suffolk County, Massachusetts
```

示例 7-25 中有两个 ORDER BY，前一个在 OVER 子句内生效，表明窗口可见区内的记录顺序，后一个针对整句生效，表明返回记录的整体顺序。请不要将二者的作用域混淆。

PARTITION BY 和 ORDER BY 可以联用，其效果就是对 PARTITION BY 指定的记录集进行排序。示例 7-26 还是复用了前面的例子，但在 OVER 子句中联用了 PARTITION BY 和 ORDER BY。

示例 7-26：联用 PARTITION BY 和 ORDER BY

```
SELECT tract_id, val,
    SUM(val) OVER (PARTITION BY left(tract_id,5) ORDER BY val) As sum_county_ordered
FROM census.facts
WHERE fact_type_id = 2
ORDER BY left(tract_id,5), val;

  tract_id   |   val    | sum_county_ordered
-------------+----------+-------------------
 25001014100 |  226.000 |            226.00
 25001011700 |  971.000 |          1197.000
 25001010208 |  984.000 |          2181.000
 :
 25003933200 |  564.000 |           564.000
 25003934200 |  593.000 |          1157.000
 25003931300 |  606.000 |          1763.000
 :
```

可以看到上面输出的合计值是逐行累加的，这就是在 OVER 子句中应用了 ORDER BY 后的效果，即窗口可见域是从排序后的记录集的头条记录开始，到 ORDER BY 字段值与当前记录值匹配的那行记录为止，因此最终会呈现为动态累加的效果。例如，对于第三个数据分区中的第五条记录来说，合计值仅会包含该分区中的前五条记录的值。在上面的示例中，我们在语句的最后加上了 ORDER BY left(tract_id,5), val 这个排序动作，因此动态累加效果一目了然。但请一定要牢记，OVER 子句中的 ORDER BY 与整句尾部的 ORDER BY 的作用是完全不同的。

你还可以通过 RANGE 或者 ROWS 关键字来显式指定窗口的可见记录域。例如：ROWS BETWEEN CURRENT ROW AND 5 FOLLOWING。

PostgreSQL 还支持建立命名窗口，该功能适用于在同一个查询中使用了多个窗口函数且每个窗口函数的窗口定义都相同的情况。示例 7-27 演示了建立命名窗口的方法，同时还展示了 LEAD 和 LAG 窗口函数的用法，这两个窗口函数可以取出当前窗口中排在当前记录行之前或者之后的记录。

示例 7-27：命名窗口以及 LEAD 和 LAG 函数的用法

```
SELECT * FROM(
    SELECT
        ROW_NUMBER() OVER( wt ) As rnum, ❶
        substring(tract_id,1, 5) As county_code,
        tract_id,
        LAG(tract_id,2) OVER wt As tract_2_before,
        LEAD(tract_id) OVER wt As tract_after
    FROM census.lu_tracts
    WINDOW wt AS (PARTITION BY substring(tract_id,1, 5) ORDER BY tract_id) ❷
) As x
WHERE rnum BETWEEN 2 and 3 AND county_code IN ('25007','25025')
ORDER BY county_code, rnum;

rnum | county_code |  tract_id   | tract_2_before | tract_after
-----+-------------+-------------+----------------+-------------
2    | 25007       | 25007200200 |                | 25007200300
3    | 25007       | 25007200300 | 25007200100    | 25007200400
2    | 25025       | 25025000201 |                | 25025000202
3    | 25025       | 25025000202 | 25025000100    | 25025000301
```

❶ 直接复用窗口名，而不需要把窗口的完整定义再输一遍。

❷ 将我们的窗口命名为 wt 窗口。

LEAD 和 LAG 函数都有一个可选的 step 实参，该实参可以是正数也可以是负数，代表需要从当前记录开始向前或者向后跳几条记录才能访问到目标记录。当 LEAD 和 LAG 在寻找目标记录的过程中跳出了当前窗口的可见域时，就会返回 NULL。这种情况是经常会遇到的。

请注意：在 PostgreSQL 中，系统自带的以及用户自定义的聚合函数都可以作为窗口函数使用，但其他数据库一般仅支持 AVG、SUM、MIN、MAX 这些系统内置聚合函数作为窗口函数使用。

7.5 CTE表达式

公用表表达式（CTE）本质上来说就是在一个非常庞大的 SQL 语句中允许用户通过一个子查询语句先定义出一个临时表，然后在这个庞大的 SQL 语句的不同地方都可以直接使用这个临时表。PostgreSQL 从 8.4 版开始支持此特性，从 9.1 版开始又扩展支持了可写 CTE，从而使得 CTE 功能得到进一步的完善。CTE 本质上就是当前语句执行期间内有效的临时表，一旦当前语句执行完毕，其内部的 CTE 表也随之失效。

有以下三种类型的 CTE。

- 基本CTE

  这是最普通的 CTE，它可以使得 SQL 语句的可读性更高，同时规划器在解析到这种 CTE 时会判定其查询代价是否很高，如果是的话，会考虑将其查询结果临时物化存储

下来（此处概念跟物化视图非常类似），这样整个 SQL 语句的其他部分再访问此 CTE 时就会更快。

- 可写CTE

  这是对基本 CTE 的一个功能扩展，其内部可以执行 UPDATE、INSERT 或者 DELETE 操作。该类 CTE 最后一般会返回修改后的记录集。

- 递归CTE

  该类 CTE 在普通 CTE 的基础上增加了一个循环操作。在执行过程中，递归 CTE 返回的结果集会有所变化。

PostgreSQL 支持可写的递归 CTE 这种复合类型。

7.5.1 基本CTE用法介绍

基本 CTE 的用法如示例 7-28 所示。WITH 关键字后跟着的就是 CTE 表达式。

示例 7-28：基本 CTE

```
WITH cte AS (
    SELECT
        tract_id, substring(tract_id,1, 5) As county_code,
        COUNT(*) OVER(PARTITION BY substring(tract_id,1, 5)) As cnt_tracts
    FROM census.lu_tracts
)
SELECT MAX(tract_id) As last_tract, county_code, cnt_tracts
FROM cte
WHERE cnt_tracts > 100
GROUP BY county_code, cnt_tracts;
```

示例 7-28 中 CTE 表达式的名称是 cte，其本体是由一个 SELECT 语句定义出来的，查询字段列表中包含 tract_id、country_code、cnt_tracts 三个列。外围的 SQL 语句会将该 CTE 作为一个临时表来使用。

单个 SQL 语句中可以创建多个 CTE，CTE 之间使用逗号分隔，所有的 CTE 表达式都要落在 WITH 子句范围内，具体语法如示例 7-29 所示。多个 CTE 表达式之间的顺序不是随便排的，排在后面的 CTE 可以引用排在前面的 CTE，但反过来不行。除了这一点以外，多个 CTE 之间的排列顺序没有别的讲究。

示例 7-29：多个 CTE 的用法

```
WITH
    cte1 AS(
        SELECT
            tract_id,
            substring(tract_id,1, 5) As county_code,
            COUNT(*) OVER (PARTITION BY substring(tract_id,1,5)) As cnt_tracts
        FROM census.lu_tracts
```

```
    ),
    cte2 AS (
        SELECT
            MAX(tract_id) As last_tract,
            county_code,
            cnt_tracts
        FROM cte1
        WHERE cnt_tracts < 8 GROUP BY county_code, cnt_tracts
    )
SELECT c.last_tract, f.fact_type_id, f.val
FROM census.facts As f INNER JOIN cte2 c ON f.tract_id = c.last_tract;
```

7.5.2 可写CTE用法介绍

可写 CTE 是从 9.1 版开始支持的一个特性，它扩展了 CTE 的功能范畴，从只读扩展为可写。我们下面使用示例 6-2 中创建的日志表来演示此功能。首先创建一个子表：

```
CREATE TABLE logs_2011_01_02 (
    PRIMARY KEY (log_id),
    CONSTRAINT chk
        CHECK (log_ts >= '2011-01-01' AND log_ts < '2011-03-01')
)
INHERITS (logs_2011);
```

在示例 7-30 中，我们将父表的部分数据迁移到子表中。父表包含了 2011 年全年的数据，子表包含了 2011 年 1 月和 2 月的数据。下面的语句中使用的 `ONLY` 关键字在 7.2.7 节中有相关介绍，而 `RETURNING` 关键字在 7.2.9 节中有相关介绍。

示例 7-30：使用可写 CTE 将数据从一个分支移动到另一个分支

```
WITH t AS (
    DELETE FROM ONLY logs_2011 WHERE log_ts < '2011-03-01' RETURNING *
)
INSERT INTO logs_2011_01_02 SELECT* FROM t;
```

7.5.3 递归CTE用法介绍

PostgreSQL 官方手册中对递归 CTE 做了最好的说明："通过新增一个可选的 `RECURSIVE` 修饰符，使 CTE 从仅仅能提供一些语法便利升华为能够实现标准 SQL 语法无法实现的功能。"递归 CTE 能够使用递归语法构造出一个表达式，这一点很有意思。递归 CTE 使用 `UNION ALL` 语法来实现每次运算过程中的运算结果的递归累积。

如果要将一个基本 CTE 转换为递归 CTE，需要在 `WITH` 后加上 `RECURSIVE` 修饰符。`WITH RECURSIVE` 后面可以附带由递归表达式和非递归表达式结合而成的语句。在大多数其他数据库中，如果要表达递归关系，并不需要显式指定 `RECURSIVE` 关键字。

递归 CTE 常用于表达树状结构。我们在"使用递归 CTE 来显示树状结构"这篇博文（http://www.postgresonline.com/journal/archives/131-Using-Recursive-Common-table-

expressions-to-represent-Tree-structures.html）中提供了一个这方面的例子。

在示例 7-31 中，我们通过查询系统 catalog 来展示数据库中的级联表关系。

示例 7-31：递归 CTE

```
WITH RECURSIVE tbls AS (
    SELECT
        c.oid As tableoid,
        n.nspname AS schemaname,
        c.relname AS tablename ❶
    FROM
        pg_class c LEFT JOIN
        pg_namespace n ON n.oid = c.relnamespace LEFT JOIN
        pg_tablespace t ON t.oid = c.reltablespace LEFT JOIN
        pg_inherits As th ON th.inhrelid = c.oid
    WHERE
        th.inhrelid IS NULL AND
        c.relkind = 'r'::"char" AND c.relhassubclass
    UNION ALL
    SELECT
        c.oid As tableoid,
        n.nspname AS schemaname,
        tbls.tablename || '->' || c.relname AS tablename  ❷ ❸
    FROM
        tbls INNER JOIN
        pg_inherits As th ON th.inhparent = tbls.tableoid INNER JOIN
        pg_class c ON th.inhrelid = c.oid LEFT JOIN
        pg_namespace n ON n.oid = c.relnamespace LEFT JOIN
    pg_tablespace t ON t.oid = c.reltablespace
)
SELECT * FROM tbls ORDER BY tablename; ❹

tableoid | schemaname |            tablename
--------+-----------+---------------------------------
3152249 | public     | logs
3152260 | public     | logs->logs_2011
3152272 | public     | logs->logs_2011->logs_2011_01_02
```

❶ 查询出所有有子表而无父表的表。

❷ 这是递归查询部分，查询出了所有位于 tbls 临时表中的表的子表。

❸ 输出时在父表名之后附加子表的名称。

❹ 输出父表和所有子表。因为语句中要求输出结果按照表名排序，而每一层级的表名是将父表的名称排在子表之前，因此排序后的效果就是子表记录紧跟在其父表记录之后输出。

7.6 LATERAL横向关联语法

LATERAL 是 9.3 版本中新支持的 ANSI SQL 标准语法。该语法的用途是：假设你需要对两张

表或者两个子查询进行关联查询操作，那么参与关联运算的双方是独立的，互相不能读取对方的数据。例如，下面的查询语句会报错，因为 L.year=2011 不是位于关联的右侧的一个列。

```
SELECT *
    FROM
        census.facts L
        INNER JOIN
        (SELECT *
            FROM census.lu_fact_types
            WHERE category =
                CASE WHEN L.yr = 2011 THEN 'Housing' ELSE category END
        ) R
    ON L.fact_type_id = R.fact_type_id;
```

加上了 LATERAL 关键字后就不会再报错：

```
SELECT * FROM census.facts L INNER JOIN LATERAL
    (SELECT * FROM census.lu_fact_types
      WHERE category = CASE WHEN L.yr = 2011 THEN 'Housing' ELSE category END) R
    ON L.fact_type_id = R.fact_type_id;
```

通过使用 LATERAL 语法可以在一个 FROM 子句中跨两个表共享多列中的数据。但有个限制就是仅支持单向共享，即右侧的表可以提取左侧表中的数据，但反过来不行。

有时候为了避免编写语法极其复杂的语句，我们也需要使用 LATERAL 语法。在示例 7-32 中，关联关系中左侧的一个列充当了右侧 generate_series 函数的一个形参：

```
CREATE TABLE interval_periods(i_type interval);
INSERT INTO interval_periods (i_type)
VALUES ('5 months'), ('132 days'), ('4862 hours');
```

示例 7-32：LATERAL 语法和 generate_series 函数的关联使用

```
SELECT i_type, dt
FROM
    interval_periods CROSS JOIN LATERAL
    generate_series('2012-01-01'::date, '2012-12-31'::date, i_type) AS dt
WHERE NOT (dt = '2012-01-01' AND i_type = '132 days'::interval);
 i_type   |           dt
------------+------------------------
 5 mons     | 2012-01-01 00:00:00-05
 5 mons     | 2012-06-01 00:00:00-04
 5 mons     | 2012-11-01 00:00:00-04
 132 days   | 2012-05-12 00:00:00-04
 132 days   | 2012-09-21 00:00:00-04
 4862:00:00 | 2012-01-01 00:00:00-05
 4862:00:00 | 2012-07-21 15:00:00-04
```

LATERAL 语法还可用于以下场景：通过关联关系左侧的数据来限制右侧的查询结果集中包含的记录数量。在示例 7-33 中，我们使用 LATERAL 语法查询出最近 100 天之内登录过我们

网站（http://www.postgresonline.com）的超级用户最近 5 次登录的时间和操作日志。本例中使用的表是在 6.1.4 节和 6.1.1 节中创建的。

示例 7-33：使用 LATERAL 语法来限制关联查询中的一方返回的记录数

```
SELECT u.user_name, l.description, l.log_ts
FROM
    super_users AS u CROSS JOIN LATERAL (
    SELECT description, log_ts
    FROM logs
    WHERE
        log_ts > CURRENT_TIMESTAMP - interval '100 days' AND
        logs.user_name = u.user_name
    ORDER BY log_ts DESC LIMIT 5
    ) AS l;
```

虽然你也可以通过窗口函数来实现相同的效果，但 LATERAL 关联执行速度更快，语法也更简洁。

在同一条 SQL 语句中可以多次使用 LATERAL 关联，当需要关联多个子查询时，甚至可以级联使用 LATERAL。在有的场景下可以省略 LATERAL 关键字，此时规划器会根据关联关系的两边交叉引用的情况来智能判断出这是一个 LATERAL 操作。但我们还是建议你，为了清晰起见，最好是显式指定 LATERAL 关键字。在不支持 LATERAL 语法的 PostgreSQL 版本上执行带横向引用的 SQL 语句当然会报错。不显式指明 LATERAL 关键字还有一个风险，就是可能会造成规划器误判，最终生成的执行计划可能完全不是你想要的。

其他数据库中也提供了横向关联的能力，但其语法不符合 ANSI SQL 规范的要求。在 Oracle 中，横向关联通过管道函数实现，在 SQL Server 中使用 CROSS APPLY 或者 OUTER APPLY 语法来实现。

第 8 章 函数编写

在大多数数据库中，你都可以把若干 SQL 语句组合在一起然后将其作为一个单元来处理。PostgreSQL 也支持这种能力。这种机制在不同数据库中的名称不一样，有的叫存储过程，有的叫用户自定义函数等名称，而 PostgreSQL 统一称之为函数。

函数不是仅仅将一堆 SQL 语句编排在一起即可，其中还需要使用过程化语言（Procedural Language，PL）来对 SQL 语句的执行过程进行控制。

在 PostgreSQL 中，你可以选择使用不同的语言来编写函数，可选择的语言有很多，其中 SQL、C、PL/pgSQL、PL/Perl 以及 PL/Python 一般都会随 PostgreSQL 安装包附带。从 PostgreSQL 9.2 版开始新增了对 PL/V8 语言的支持，通过它你可以使用 JavaScript 语言来编写函数。对于 Web 应用的开发人员来说，引入对 PL/V8 的支持是很令人激动的一件事情，因为 JSON、JSONB 数据类型和 JavaScript 语言是绝配。内置的 `json` 和 `jsonb` 数据类型在 5.6 节中进行过介绍。

你可以按需安装 PL/R、PL/Java、PL/sh、PL/TSQL 等语言扩展，此外还有一些用于高端逻辑处理以及人工智能处理的试验性语言，比如 PL/Scheme 和 PL/OpenCL 等。你可以在官方手册的“过程化语言”这一节（http://www.postgresql.org/docs/current/interactive/external-pl.html）中查到 PostgreSQL 支持的完整语言列表。

8.1 PostgreSQL函数功能剖析

8.1.1 函数功能基础知识介绍

在 PostgreSQL 中，不管你选择使用何种编程语言，所编写出来的函数的结构都是类似的，如示例 8-1 所示。

示例 8-1：函数的基本结构

```
CREATE OR REPLACE FUNCTION func_name(arg1 arg1_datatype DEFAULT arg1_default)
RETURNS some type | set of some type | TABLE (..) AS
$$
BODY of function
$$
LANGUAGE language_of_function
```

函数的实参列表中可以只有实参类型而没有实参名，但如果不写实参名的话，你就不能使用类似 `arg1:=...` 这种语法来为实参赋值。实参是可选的，也就是说调用函数时可以不填实参值，如果不填的话系统就会自动为其赋一个默认值。设计函数时，在其实参列表中应将可选实参排在必选实参之后。

定义函数时可以添加一些标记符来优化执行效率或者提升安全性，支持的标记符列表如下。

- `LANGUAGE`（使用的编程语言）

  指明本函数使用的编程语言，当然该语言必须在当前函数所在的 database 中已安装。执行 `SELECT lanname FROM pg_language;` 即可查到已安装的语言列表。

- `VOLATILITY`（结果的稳定性）

  该标记符可以告诉查询规划器当该函数执行完毕后得到的结果是否可以缓存下来以供下次使用。它有以下几个可选值。

  – `IMMUTABLE`（结果恒定不变）

  任何情况下，只要调用该函数时使用相同的输入就总会得到相同的输出。也就是说该函数的内部逻辑对外界完全无依赖。

  – `STABLE`（结果相对稳定）

  如果在同一个查询语句中多次调用该函数，则每次调用时只要使用相同的输入就总会得到相同的输出。也就是说该函数的内部逻辑在当前 SQL 的上下文环境内是有恒定输出的。

  – `VOLATILE`（结果不稳定）

  每次调用该函数得到的结果可能都不同，即便每次都使用相同的输入也是这样。那些更改数据的函数或者那些依赖于比如系统时间这类环境设置的函数应该被标记为 `VOLATILE`。该项也是默认值。

请注意，VOLATILITY 标记符仅仅是给规划器提供了一个提示信息，规划器并不一定会按照此设置来进行处理。如果函数被标记为 VOLATILE，那么规划器每次遇到此函数都会重新解析并重新执行一遍；如果被标记为别的类型，那么规划器也有可能不会对其执行结果进行缓存，因为规划器可能认为重新计算一遍反而会更快。

– STRICT（严格模式）

对于一个严格模式的函数来说，如果有任何输入为 NULL，则规划器根本不会执行这个函数就直接返回 NULL。如果未显式指定为 STRICT 模式，则函数默认都是非严格模式的。写函数时对 STRICT 限定符请务必慎用，因为用了以后可能会导致规划器不使用索引。请参考我们的博文“SQL 函数的严格模式”（http://www.postgresonline.com/journal/archives/163-STRICT-on-SQL-Function-Breaks-In-lining-Gotcha.html）以获取更多细节。

- COST（执行成本估计）

  这是标记函数中计算操作密集程度的一个相对度量值。如果使用的是 SQL 或 PL/pgSQL 语言，则该值为 100；如果使用 C 语言，则该值为 1。该值会影响到规划器执行 WHERE 子句中的函数时的优先级，也会影响到是否对此函数进行结果集缓存的可能性判定。此值越大，则规划器会认为执行该函数需要耗费的时间越多。

- ROWS（返回结果集的行数估计）

  仅当函数返回的是一个结果集时此标记符才有用。该值是返回的结果集中记录数的一个估计值。规划器会利用此数值来为此函数分析得出最佳的执行策略。

- SECURITY DEFINER（安全控制符）

  如果设置了安全控制符，则会以创建此函数的用户的权限执行此函数；如果未设置，则会以调用此函数的用户的权限执行此函数。如果某用户对某张表没有操作权限而又需要操作该表，那么就可以让创建该表的用户提供一个带 SECURITY DEFINER 标识的函数来对此表进行操作。可以看出，当需要进行表的访问权控制时，这个安全控制符还是很有用的。

8.1.2 触发器和触发器函数

任何一个功能健全的数据库都支持触发器功能。借助触发器机制可以实现自动捕捉数据变化事件并进行相应处理。PostgreSQL 既支持对表建触发器也支持对视图建触发器。

可以指定触发器在语句级或者记录级被触发。对语句级触发器来说，每执行一条 SQL 语句只会被触发一次；对记录级触发器来说，SQL 语句执行过程中每修改一条记录就会被触发一次。例如，假设你对某表执行了一个 UPDATE 语句，更新了 1500 条记录。那么该表上的语句级触发器只会触发一次，而记录级触发器会触发 1500 次。

你还可以更加精细化设置触发器的触发时机，系统支持 BEFORE、AFTER 以及 INSTEAD OF 这

三种时机。BEFORE 类的触发器会在语句执行之前或者记录行被修改之前触发，你可以借此时机来取消此次修改或者对要修改的数据进行预先备份。AFTER 类的触发器会在语句执行之后或者记录行被修改之后触发，你可以借此时机来获得修改后的新值，该类触发器一般用于记录修改日志或者进行数据复制。INSTEAD OF 类的触发器会将原语句的操作内容替换掉。BEFORE 和 AFTER 类型的触发器只能用于表，而 INSTEAD OF 类触发器只能用于视图。

你还可以在定义触发器时加上 WHEN 条件来限定只有那些满足了筛选条件的记录被修改时才激活该触发器；也可以通过加上“UPDATE OF+ 字段列表”子句来指定只有修改了特定的列时才激活该触发器。如果希望更加深入细致地了解触发器与主体语句之间的触发联动机制，请参考 PostgreSQL 官方手册中的“触发器行为概览”（http://www.postgresql.org/docs/current/interactive/trigger-definition.html）一节的内容。我们还在示例 7-4 中演示了一个视图触发器的用法。

PostgreSQL 提供了一种专门用于处理触发器逻辑的函数，这类函数被称为触发器函数，其行为模式与其他函数完全类似，内部的代码结构也相同。触发器函数与普通函数的唯一区别在于输入形参和输出类型。触发器函数从不需要实参，因为可以在函数内部访问数据并对其进行修改。

触发器函数的返回值永远是 trigger 类型。PostgreSQL 的触发器函数与别的普通函数机制完全类似，因此一个触发器函数可以被多个触发器公用。几乎没有哪家数据库能支持该特性，因为一般数据库中都是把触发器和触发器函数作为一个完整的对象绑定在一起的，这样的触发器处理逻辑无法被别的触发器重用。

在 PostgreSQL 中，每个触发器有且仅有一个配套的触发器函数。如果由于业务需要必须将逻辑分散到多个触发器函数中，那么就得创建多个触发器来调用它们，这些触发器的触发事件可以相同也可以不同，如果触发事件相同的话，那么系统会将触发器名称按字典顺序进行排序，然后逐个触发。后面一个触发器可以看到前面一个触发器的修改结果。每一个触发器并不是一个独立的事务，因此如果在某个触发器中执行了回滚操作，那么在此触发器之前执行过的触发器修改都会被回滚掉。

你可以使用 PostgreSQL 支持的任何一种编程语言来编写触发器函数，但请注意不能使用 SQL，因为它不是过程式语言。SQL 对应的过程式语言是 PL/pgSQL，它也是目前为止在 PostgreSQL 环境中使用最广泛的语言。8.3.2 节会介绍相关的用法。

8.1.3 聚合操作

大多数其他数据库仅允许使用 ANSI SQL 标准中定义的那些聚合函数，比如 MIN、MAX、AVG、SUM 以及 COUNT 等。在 PostgreSQL 中则无此限制，你可以自行实现比以上函数功能更复杂的聚合函数。在 PostgreSQL 中，一个聚合函数同时也可以作为窗口函数（相关概念请参见 7.4 节）来使用，因此你可以实现事半功倍的效果。

你可以使用 PostgreSQL 所支持的包括 SQL 语言在内的几乎任何语言来编写聚合函数。一个聚合函数一般是基于一个或者多个子函数实现的。首先至少得有一个状态转换函数，该函数会反复执行多次以将输入的多行记录聚合为一个单独的结果。你还可以建立用于处理初始状态和终结状态的函数，不过这两个函数是可选的。前述这几类函数都是聚合函数的子函数，它们可以使用不同的编程语言来实现，我们在“PostgreSQL 的聚合函数”（http://www.postgresonline.com/journal/index.php?/plugin/tag/aggregates）这篇博文中演示了基于 PL/pgSQL、PL/Python 和 SQL 等多种语言的子函数构造而成的聚合函数实例。

不管你使用何种编程语言来编写这些子函数，最终将它们整合为一个聚合函数的语法是一样的，如下所示。

```
CREATE AGGREGATE my_agg (input data type) (
SFUNC=state function name,
STYPE=state type,
FINALFUNC=final function name,
INITCOND=initial state value, SORTOP=sort_operator
);
```

SFUNC 状态切换函数（这个名称不够直观，此处所谓的“状态”是指在聚合运算过程中每处理完一条记录后得到的中间结果）是实现聚合运算的逻辑主体，它会将自身上一次被调用后生成的计算结果作为本次计算的输入，同时输入的还有当前新一条的待处理记录，这样将所有记录一条条累积处理完毕后，就得到了基于整个目标记录集的“状态”，也就是最终的聚合结果。有的情况下，SFUNC 处理得到的结果就是聚合函数需要的最终结果，但另外一些情况下 SFUNC 处理完毕的结果还需要再进行最终加工才是我们想要的聚合结果，FINALFUNC 就是负责这个最终加工步骤的函数。FINALFUNC 是可选的，由于它的作用是对 SFUNC 函数的输出结果做最后加工，因此该函数的输入一定是 SFUNC 函数的输出。INITCOND 也是可选的，如果设定了该条目，那么其值会被作为 SFUNC 函数的“状态”的初始值。

最后的 SORTOP 也是可选的，其值是类似于 > 或 < 这样的运算符，它的作用是为类似 MAX、MIN 这样的排序操作指定排序运算符。指定了 SORTOP 运算符后，规划器会使用索引来进行 MAX、MIN 这样的聚合运算，由于索引是有序的，所以可以快速定位到索引的头部或者尾部寻找 MAX、MIN 值，这样就不需要对所有记录逐条进行大小值判断，整体运算速度就得以极大提升。不过 SORTOP 运算符的使用有一个先决条件，那就是在聚合运算的目标表上，以下两条语句的执行结果必须完全相同。

```
SELECT agg(col) FROM sometable;

SELECT col FROM sometable ORDER BY col USING sortop LIMIT 1;
```



在 PostgreSQL 9.4 版中，`CREATE AGGREGATE` 语法得到了强化，新增了对移动窗口聚合函数的支持，该特性对于窗口可移动的窗口函数是很有意义的。更多详情请参考 9.4 版官方手册中的“创建聚合函数”这一节（http://www.postgresql.org/docs/9.4/interactive/sql-createaggregate.html）的内容。

一般来说聚合函数只针对一个列进行聚合运算，比如 `MAX`、`MIN`、`AVG` 等，但事实上完全可以建立针对多个列进行聚合运算的聚合函数。如果你需要这样的多列聚合函数，请参考“如何创建基于多个列的聚合函数”（http://www.postgresonline.com/journal/archives/105-How-to-create-multi-column-aggregates.html）这篇博文来了解具体的实现方法。

前面已经介绍过聚合函数是可以使用 SQL 来编写的。SQL 是一种极其易用的语言，你不需要关心那些各式各样的流程控制语句（因为 SQL 中不支持），而且你也很有可能对 SQL 已经很熟悉，因此上手更加简单。当编写聚合函数时，仅仅使用 SQL 就可以实现很强大的功能。我们将在 8.2.2 节中介绍相关内容。

8.1.4　受信与非受信语言

PostgreSQL 支持的函数语言可按照信任级别分为两类：受信语言与非受信语言。很多语言（但并不是所有）同时提供了受信与非受信版本。这里所说的“受信”是指该语言不可能对数据库服务器的底层操作系统造成任何破坏。

- *受信语言*

  受信语言不具备直接访问数据库服务器底层文件系统的权限，因此在该类语言中不能直接执行操作系统级命令。任何权限级别的用户都可以使用受信语言创建函数。包括 SQL、PL/pgSQL、PL/Perl 在内的语言都是受信语言。

- *非受信语言*

  非受信语言可以直接与操作系统进行交互，通过该类语言可以直接调用操作系统提供的函数和 Web 服务接口。PostgreSQL 中只有超级用户才有权使用非受信语言编写函数，但超级用户有权将基于非受信语言的函数的执行权限授予普通用户。一般来说，非受信语言的命名会以 U 结尾，比如 PL/PerlU、PL/PythonU 等。

8.2　使用SQL语言来编写函数

大多数情况下我们仅仅使用 SQL 语言来编写单个语句，但事实上它也可以用于编写函数。在 PostgreSQL 中，用 SQL 编写函数是一件很简单的事情：只需在现成的 SQL 基础上加上函数头和函数尾就可以了。但编写简单同时也意味着功能有限。SQL 不是一种过程式语言，因此你就无法用上比如条件分支判断这种过程控制语句。此外还有一个更严重的限制，那就是无法运行根据传入到函数中的实参即时组合成的动态 SQL 语句。

当然，SQL 函数也有其优点。查询规划器可以深入到 SQL 函数内部并对其中每一条 SQL 语句进行分析和优化，该过程被称为 inlining，即内联处理。规划器对于别的语言编写的函数只能当成黑盒处理。内联处理机制使得 SQL 函数能够充分利用索引从而提高执行效率。

8.2.1 编写基本的SQL函数

示例 8-2 演示了一个最基本的 SQL 函数，该函数向表中插入一条记录，并返回一个标量值。

示例 8-2：创建一个 SQL 函数，其返回值为新插入的记录的唯一 ID

```
CREATE OR REPLACE FUNCTION write_to_log(param_user_name varchar, param_description
text)
RETURNS integer AS
$$
INSERT INTO logs(user_name, description) VALUES($1, $2)
RETURNING log_id;
$$
LANGUAGE 'sql' VOLATILE;
```

函数的调用语法如下所示。

```
SELECT write_to_log('alejandro', 'Woke up at noon.') As new_id;
```

类似地，也可以在 SQL 函数中更新数据并返回一个标量或者不返回，如示例 8-3 所示。

示例 8-3：创建一个进行更新操作的 SQL 函数

```
CREATE OR REPLACE FUNCTION
update_logs(log_id int, param_user_name varchar, param_description text)
RETURNS void AS
$$
UPDATE logs SET user_name = $2, description = $3
 , log_ts = CURRENT_TIMESTAMP WHERE log_id = $1;
$$
LANGUAGE 'sql' VOLATILE;
```

通过以下语句来调用此函数。

```
SELECT update_logs(12, 'alejandro', 'Fell back asleep.');
```



在 9.2 版之前，SQL 函数仅可以在函数主体中使用输入实参的序号位置。从 9.2 版开始支持使用命名实参的选项。比如，你可以使用 `param_1` 和 `param_2` 来代替之前 `$1` 和 `$2` 这种写法。除了 SQL 以外的其他语言在 9.2 版之前无此限制，也就是说它们一直都可以使用实参名来引用实参。

基本上所有编程语言编写的函数都支持返回结果集，SQL 函数也不例外，它有三种返回结果集的方法：第一种是 ANSI SQL 标准中规定的 `RETURNS TABLE` 语法，第二种是使用 `OUT` 形参，第三种是使用复合数据类型。第一种的 `RETURNS TABLE` 语法是从 PostgreSQL 8.3 版

才开始支持的，其他数据库一般也都是基于该语法来实现结果集的返回。在示例 8-4 中，我们演示了如何使用这三种方法来实现返回结果集。

示例 8-4：在函数中返回结果集

使用 RETURNS TABLE 语法的方式如下所示。

```
CREATE OR REPLACE FUNCTION select_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text, log_ts time
stamptz) AS
$$
SELECT log_id, user_name, description, log_ts FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

使用 OUT 形参的方式如下所示。

```
CREATE OR REPLACE FUNCTION select_logs_out(param_user_name varchar, OUT log_id int
 , OUT user_name varchar, OUT description text, OUT log_ts timestamptz)
RETURNS SETOF record AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

使用复合数据类型的方式如下所示。

```
CREATE OR REPLACE FUNCTION select_logs_so(param_user_name varchar)
RETURNS SETOF logs AS
$$
SELECT * FROM logs WHERE user_name = $1;
$$
LANGUAGE 'sql' STABLE;
```

以上三种方式实现的函数的调用方法都是一样的。

```
SELECT * FROM select_logs_xxx('alejandro');
```

8.2.2 使用SQL语言编写聚合函数

本节将演示如何使用 SQL 语言来创建一个用于计算几何平均值的聚合函数。几何平均值（http://www.buzzardsbay.org/geomean.htm）是指 *n* 个正数的连乘积的 *n* 次方根（`(x1*x2*x3...xn)^(1/n)`），它在金融、经济以及统计学领域有着广泛的应用。当样本数字的值域范围变化很大时，可以使用几何平均值来替代更常见的算术平均数。几何平均值可以使用更高效的公式来计算：`EXP(SUM(LN(x))/n)`，该公式使用了对数来将连续的乘法运算转换为连续的加法运算，因此计算机执行的效率更高。在下面的例子中，我们将使用该公式计算几何平均值。

为了实现几何平均值运算，我们使用了两个子函数：一个状态转换函数，用于把对数运算

结果相加（参见示例 8-5）；一个最终处理函数，用于对对数之和进行取幂运算。此外我们还需要指定状态初始值为 0。

示例 8-5：创建几何平均值聚合函数的状态切换函数

```
CREATE OR REPLACE FUNCTION geom_mean_state(prev numeric[2], next numeric)
RETURNS numeric[2] AS
$$
SELECT
  CASE
    WHEN $2 IS NULL OR $2 = 0 THEN $1
    ELSE ARRAY[COALESCE($1[1],0) + ln($2), $1[2] + 1]
  END;
$$
LANGUAGE sql IMMUTABLE;
```

此处定义的状态切换函数有两个输入项：第一个是前次调用本状态切换函数计算后得到的结果，其类型为含两个元素的数字型数组；第二个是本轮计算要处理的样本值。如果第二个实参的值为 NULL 或者为 0，则本轮无需计算，直接返回实参 1 的值；否则将本次处理的样本数字的 ln 对数值累加到实参数组的第一个元素上，并对实参数组的第二个元素值加 1。这样最终得到结果就是含所有样本数字的 ln 对数值的总和以及总运算次数。

此外我们还需要一个如示例 8-6 所示的最终处理函数，该函数中需要将状态转换函数计算得到的两个值相除。

示例 8-6：创建几何平均值聚合函数的最终处理函数

```
CREATE OR REPLACE FUNCTION geom_mean_final(numeric[2])
RETURNS numeric AS
$$
SELECT CASE WHEN $1[2] > 0 THEN exp($1[1]/$1[2]) ELSE 0 END;
$$
LANGUAGE sql IMMUTABLE;
```

最后，我们需要将前面定义的这些子函数整合到一起组成一个完整的聚合函数，语法如示例 8-7 所示。（请注意，本例中的聚合运算需要一个初始值 (`0,0`)，该初始值的类型与 `SFUNC` 的实参类型一定是一致的。）

示例 8-7：基于定义好的子函数来创建几何平均值聚合函数

```
CREATE AGGREGATE geom_mean(numeric) (
SFUNC=geom_mean_state,
STYPE=numeric[],
FINALFUNC=geom_mean_final,
INITCOND='{0,0}'
);
```

接下来我们测试一下刚刚创建好的 `geom_mean` 聚合函数。在示例 8-8 中，我们计算出了马萨诸塞州各县的种族多样性排名，并列出种族多样性最好的 5 个县的数据。

示例 8-8：基于几何平均值来统计出种族多样性最好的 5 个县

```
SELECT left(tract_id,5) As county, geom_mean(val) As div_county
FROM census.vw_facts
WHERE category = 'Population' AND short_name != 'white_alone'
GROUP BY county
ORDER BY div_county DESC LIMIT 5;

county |     div_county
-------+---------------------
25025  | 85.1549046212833364
25013  | 79.5972921427888918
25017  | 74.7697097102419689
25021  | 73.8824162064128504
25027  | 73.5955049035237656
```

接下来我们大胆一点，直接将上面定义的聚合函数当作窗口函数来试一下，看效果如何，如示例 8-9 所示。

示例 8-9：列出 5 个种族多样性最好的人口普查区

```
WITH X AS (SELECT
  tract_id,
  left(tract_id,5) As county,
  geom_mean(val) OVER (PARTITION BY tract_id) As div_tract,
  ROW_NUMBER() OVER (PARTITION BY tract_id) As rn,
  geom_mean(val) OVER(PARTITION BY left(tract_id,5)) As div_county
FROM census.vw_facts WHERE category = 'Population' AND short_name != 'white_alone'
)
SELECT tract_id, county, div_tract, div_county
FROM X
WHERE rn = 1
ORDER BY div_tract DESC, div_county DESC LIMIT 5;

tract_id    | county |      div_tract       |     div_county
------------+--------+----------------------+------------------
25025160101 | 25025  | 302.6815688785928786 | 85.1549046212833364
25027731900 | 25027  | 265.6136902148147729 | 73.5955049035237656
25021416200 | 25021  | 261.9351057509603296 | 73.8824162064128504
25025130406 | 25025  | 260.3241378371627137 | 85.1549046212833364
25017342500 | 25017  | 257.4671462282508267 | 74.7697097102419689
```

8.3 使用PL/pgSQL语言编写函数

如果 SQL 语言已经不能满足你编写函数的需求，一般来说常见的解决方案是转为使用 PL/pgSQL。PL/pgSQL 优于 SQL 之处在于它支持通过 `DECLARE` 语法定义本地变量以及支持流程控制语法。

8.3.1 编写基础的PL/pgSQL函数

为了向你展示 PL/pgSQL 与 SQL 之间的语法区别，我们在示例 8-10 中用 PL/pgSQL 重写了

示例 8-4 中的函数例子。

示例 8-10：使用 PL/pgSQL 编写返回值为表类型的函数

```
CREATE FUNCTION select_logs_rt(param_user_name varchar)
RETURNS TABLE (log_id int, user_name varchar(50), description text, log_ts time
stamptz) AS
$$
BEGIN RETURN QUERY
    SELECT log_id, user_name, description, log_ts FROM logs
     WHERE user_name = param_user_name;
END;
$$
LANGUAGE 'plpgsql' STABLE;
```

8.3.2 使用PL/pgSQL编写触发器函数

由于 PostgreSQL 不支持使用 SQL 编写触发器函数，因此 PL/pgSQL 就成了编写触发器函数的首选。本节中，我们将向你介绍如何使用 PL/pgSQL 编写基本的触发器函数。

总共需要两个步骤：第一步是写一个触发器函数，第二步是将此触发器函数显式附加到合适的触发器上。这第二步将处理触发器的函数与触发器本身分离开，这是 PostgreSQL 的一个强大功能。你可以将同一个触发器函数附加到多个触发器上，从而实现触发器函数逻辑的重用。该模式是 PostgreSQL 的独创功能，没有任何别的数据库能支持该特性。由于触发器函数之间是完全独立的，因此你可以为每个触发器函数选择不同的编程语言，这些不同语言编写的触发器完全可以协同工作。PostgreSQL 支持通过一个触发事件（`INSERT`、`UPDATE`、`DELETE`）激活多个触发器，而且每个触发器可以基于不同语言编写。例如，假设数据库中发生某一事件时你需要将其记录下来，另外还需要发邮件通知你。那么你可以使用 PL/PythonU 或者 PL/PerlU 语言编写一个具备发送邮件功能的触发器；同时可以使用 PL/pgSQL 语言编写一个记录日志的触发器。发生指定事件时，这两个触发器会同时被触发，各自执行各自的任务。

示例 8-11 中演示了如何创建一个基本的触发器函数及配套的触发器。

示例 8-11：通过触发器对新插入的记录或者修改的记录打时间戳

```
CREATE OR REPLACE FUNCTION trig_time_stamper() RETURNS trigger AS ❶
$$
BEGIN
    NEW.upd_ts := CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql VOLATILE;

CREATE TRIGGER trig_1
BEFORE INSERT OR UPDATE OF session_state, session_id ❷
ON web_sessions
FOR EACH ROW EXECUTE PROCEDURE trig_time_stamper();
```

❶ 定义触发器函数。该函数适用于任何带有 upd_ts 字段的表，该函数会先将 upd_ts 字段的值更新为当前时间戳，然后再返回修改后的记录。这种修改记录新值的动作应该放在 BEFORE 触发器中，AFTER 触发器被触发时记录新值已经写入表中，因此时机已经错过，此时所有对记录新值的修改动作都会被忽略。

❷ "字段级触发"是 9.0 版开始支持的一个特性，通过该特性可以将触发器的触发时机精确到字段级别。在 9.0 版之前，只要发生了 UPDATE 或者 INSERT 动作，上面示例中的触发器都会被触发；因此如果要实现字段级触发控制就必须拿 OLD.some_column 和 NEW.some_column 进行对比，找到发生变化的字段，然后才能判定是否要进行"字段级触发"。（请注意：INSTEAD OF 触发器不支持该特性。）

8.4 使用PL/Python语言编写函数

Python 是一种非常灵活的语言，它支持非常丰富的功能扩展库。据我们所知，PostgreSQL 是唯一一种允许用户使用 Python 语言来编写函数的数据库。从 9.0 版开始，PostgreSQL 还同时支持了 Python 2 和 Python 3 两种语言。



你可以在同一个 database 中同时安装 PL/Python2U 和 PL/Python3U 这两个语言包，但在同一个用户会话上不能同时使用这两种语言。这就意味着你不能在同一个语句中同时调用分别由 PL/Python2U 和 PL/Python3U 编写的函数。你在系统中会见到一种叫作 PL/PythonU 的语言，它实际上是系统为了保持前向兼容而为 PL/Python2U 语言建的一个别名。

在使用 PL/Python 语言之前，要先在服务器上搭建好 Python 运行环境。Windows 和 Mac 平台的 Python 安装包可以从 http://www.python.org/download/ 站点下载到。Linux/Unix 平台的各种发行版上一般都已经附带了 Python 环境，因此无需额外安装。请参考 PostgreSQL 官方手册中对 PL/Python 的相关介绍（http://www.postgresql.org/docs/current/interactive/plpython.html）来了解详情。搭建好 Python 运行环境之后，需要为 PostgreSQL 安装 Python 语言扩展包。

```
CREATE EXTENSION plpython2u;
CREATE EXTENSION plpython3u;
```

在 PostgreSQL 上安装 Python 语言扩展包之前，请务必确保服务器操作系统上的 Python 运行环境已经正常，否则你可能会遇到各种奇奇怪怪的问题。

由于 PostgreSQL 的 PL/PythonU 语言扩展包是基于某个具体版本的 Python 语言包编译出来的，因此你需要保证服务器上的 Python 版本与 plpythonu 扩展包的版本是匹配的。例如，假设你的 plpython2u 扩展包是基于 Python 2.7 版编译的，那么服务器上就需要安装好 Python 2.7 运行环境。

编写基本的Python函数

PostgreSQL 会自动在 PostgreSQL 数据类型与 Python 数据类型间进行双向转换。PL/Python 语言编写的函数支持返回数组和复合数据类型。你可以使用 PL/Python 来编写触发器函数和聚合函数。我们在 Postgres Online 站点上提供了一系列介绍 PL/Python 的文章（http://www.postgresonline.com/journal/index.php?/plugin/tag/plpython），其中有相关的语法示例。

Python 语言可以实现一些通过 PL/pgSQL 语言无法实现的功能。在示例 8-12 中，我们演示了如何使用 PL/Python 语言来编写一个文本搜索函数，该函数可以实现对 PostgreSQL 在线官方手册的内容进行检索。

示例 8-12：使用 PL/Python 语言编写的函数来搜索 PostgreSQL 官方手册的内容

```
CREATE OR REPLACE FUNCTION postgresql_help_search(param_search text)
RETURNS text AS
$$
import urllib, re ❶
response = urllib.urlopen(
  'http://www.postgresql.org/search/?u=%2Fdocs%2Fcurrent%2F&q=' + param_search
) ❷
raw_html = response.read() ❸
result = raw_html[raw_html.find("<!-- docbot goes here -->") : raw_html.find("<!--
pgContentWrap -->") - 1] ❹
result = re.sub('<[^<]+?>', '', result).strip() ❺
return result ❻
$$
LANGUAGE plpython2u SECURITY DEFINER STABLE;
```

❶ 导入我们接下来需要使用的功能库。

❷ 在连接搜索词之后执行搜索。

❸ 读取返回的搜索结果并将其保存到一个名为 `raw_html` 的变量中。

❹ 从 `raw_html` 中将 `<!-- docbot goes here -->` 和 `<!-- pgContentWrap -->` 之间包含的内容截取出来并存放到一个名为 `result` 的新变量中。

❺ 将 `result` 开头部分和结尾部分的 HTML 标记和空格删除掉。

❻ 返回 `result` 变量的内容。

调用 Python 函数与调用别的语言编写的函数没什么两样。在示例 8-13 中，我们使用在示例 8-12 中创建的函数来搜索三个字符串。

示例 8-13：在查询语句中使用 Python 函数

```
SELECT search_term, left(postgresql_help_search(search_term), 125) As result
FROM (VALUES ('regexp_match'),('pg_trgm'),('tsvector')) As x(search_term);
```

前面提到过 PL/Python 是一种非受信语言，而且没有相应的受信版本。这意味着只有超级用户才能使用 PL/Python 编写函数，并且使用该语言编写出来的函数可以直接操作文件系

统。示例 8-14 就利用了 PL/Python 的这种能力来获得一个目录中的文件列表。请注意，从操作系统的角度来看，PL/Python 函数是以 PostgreSQL 安装时创建的 `postgres` 操作系统账户身份来执行的，因此你在执行该示例之前需要确保 `postgres` 账户对该示例中使用的目录拥有访问权限。

示例 8-14：列出一个目录中的所有文件

```
CREATE OR REPLACE FUNCTION list_incoming_files()
RETURNS SETOF text AS
$$
import os
return os.listdir('/incoming')
$$
LANGUAGE 'plpython2u' VOLATILE SECURITY DEFINER;
```

可以通过以下语句执行上面创建的函数。

```
SELECT filename
FROM list_incoming_files() As filename
WHERE filename ILIKE '%.csv'
```

8.5 使用PL/V8、PL/CoffeeScript以及PL/LiveScript语言来编写函数

PL/V8（http://code.google.com/p/plv8js/wiki/PLV8，又名 PL/JavaScript）是一种基于 Google V8（http://code.google.com/p/v8/）引擎的受信语言。通过它可以实现用 JavaScript 来编写函数并使用 JSON 数据类型来与外界交互。PL/V8 并不是 PostgreSQL 的一个核心功能，因此在比较流行的 PostgreSQL 发行版中一般都不附带此语言包，只有 Heroku 是个例外。你可以通过源码自行编译安装。我们已经为你提供了编译好的 Windows 平台安装包，适用于 PostgreSQL 9.2 和 9.3 版。9.2 版的下载地址是（含 32 位和 64 位版本）：http://www.postgresonline.com/journal/archives/280-PLV8-1.3-windows-binaries-for-PostgreSQL-9.2.html。9.3 版的下载地址是（含 32 位和 64 位版本）：http://www.postgresonline.com/journal/archives/305-PostgreSQL-9.3-extension-treats-for-windows-users-plV8.html。

尽管 PostgreSQL 9.1 版已经支持 PL/V8，但我们还是强烈建议你升级到 9.2 版，因为该版本开始原生支持 JSON 数据类型。

在 PostgreSQL 中安装了 PL/V8 扩展包后，你会发现新增支持的语言不是一种，而是三种，不过它们都是 JavaScript 的相关语言。

- `PL/V8（plv8）`

  这是最基本的 JavaScript 语言，也是下面两种语言的基础。

- `PL/CoffeeScript`（`plcoffee`）

  CoffeeScript 是一门简洁的、构架于 JavaScript 之上的预处理器语言，可以静态编译成 JavaScript。其语法类似于 Python，也使用了缩进格式来表达代码段之间的隶属关系，从而省掉了烦人的大括号。

- `PL/LiveScript`（`plls`）

  LiveScript 是 CoffeeScript 语言的一个分支，其语法与 CoffeeScript 类似，但拥有更多的语法特性。"从 CoffeeScript 转到 LiveScript 的 10 个理由"这篇文章（http://livescript.net/blog/ten-reasons-to-switch-from-coffeescript.html）中认为 LiveScript 是 CoffeeScript 的理想替代品。相比 CoffeeScript 而言，LiveScript 拥有更多类似于 Python、F# 和 Haskell 的特性。如果你正在寻找一门比 PL/Python 占用内存空间更小的受信语言，那么应该试试 LiveScript。

PL/CoffeeScript 和 PL/LiveScript 语言都是基于相同版本的 PL/V8 库编译的，因此二者的功能本质上与 PL/V8 完全一致。事实上，如果这两种语言你试用之后觉得不合适，那么也可以很轻易地切换回 PL/V8。这三种语言都是受信语言，这意味着它们无法访问底层文件系统，但没有超级用户权限的用户们可以用它们来实现函数。

示例 8-15 中是创建这三个语言包的命令。如果你需要在某个 database 中使用这些语言，那么就必须在其中执行一遍这些安装命令。这三种语言可以单独按需安装。

示例 8-15：PL/V8 系列语言包的安装

```
CREATE EXTENSION plv8;
CREATE EXTENSION plcoffee;
CREATE EXTENSION plls;
```

与 PL/pgSQL 语言相比，上述的 PL/V8 系列语言能够提供很多关键的功能特性，这使得它们有着独特的存在价值。这些特性中的一部分只有 PL/R 这种高端过程化语言才能支持。

- 与 SQL 和 PL/pgSQL 相比，数学运算速度更快。
- 创建窗口函数的能力。SQL、PL/pgSQL、PL/Python 均不支持该能力（但 PL/R 和 C 语言是支持的）。
- 创建触发器函数和聚合函数的能力。
- 支持语句预解析、子事务、内嵌函数、类以及 try-catch 异常处理机制。
- 使用 `eval` 函数动态执行 JavaScript 代码的能力。
- 支持 JSON 数据类型，能对 JSON 对象进行循环筛选处理。
- 在 `DO` 命令的匿名代码块中访问函数的能力。
- PL/V8 和 Node.js（http://nodejs.org/）均使用了谷歌 V8 引擎，因此很多适用于 Node.js 的库可以不经修改就直接应用于 PL/V8。这给 Node.js 的开发人员以及其他使用 JavaScript 进行网络应用开发的人们带来了很多便利。PostgreSQL 中有一个名为 `plv8x`

（https://github.com/clkao/plv8x）的扩展包，该扩展包使得使用 Node.js 模块和你构建的模块在 PL/V8 中重用更加容易。

我们的 PostgresOnline 博客站点上提供了一些介绍 PL/V8 用法的例子。在有的例子里面，我们从网上找来了大块的 JavaScript 代码并修改移植为 PL/V8 语言，详情可参考“使用 PLV8 语言来构造 JSON 查询器”（http://www.postgresonline.com/journal/archives/272-Using-PLV8-to-build-JSON-selectors.html）这篇博文。前述 PL/V8 系列语言可完美地辅助 Web 应用开发，因为大量的客户端 JavaScript 代码都是可以直接拿来重用的。不过，对于 PostgreSQL 来说，这几种语言更为重要的意义在于它们都是全功能的强大语言，可用于处理数值计算、数据修改以及很多其他任务。

8.5.1 编写基本的函数

PL/V8 语言的主要优点之一就是可以在 PL/V8 函数中直接调用任何 JavaScript 函数，而且几乎不需要做修改。例如，网上有很多对电子邮件地址进行合法性验证的 JavaScript 函数，我们随便找了一个并将其封装为 PL/V8 函数，如示例 8-16 所示。

示例 8-16：使用 PL/V8 函数来验证电子邮件地址的合法性

```
CREATE OR REPLACE FUNCTION
validate_email(email text) returns boolean as
$$
 var re = /\S+@\S+\.\S+/;
 return re.test(email);
$$ LANGUAGE plv8 IMMUTABLE STRICT;
```

上面的例子中使用了一个 JavaScript 正则表达式对象来检查电子邮件地址的合法性。示例 8-17 演示了如何使用此函数。

示例 8-17：调用 PL/V8 语言编写的电子邮件地址合法性校验函数

```
SELECT email, validate_email(email) AS is_valid
 FROM (VALUES ('alexgomezq@gmail.com')
 ,('alexgomezqgmail.com'),('alexgomezq@gmailcom')) AS  x (email);
```

输出结果如下：

```
       email         | is_valid
---------------------+------------
alexgomezq@gmail.com | t
alexgomezqgmail.com  | f
alexgomezq@gmailcom  | f
```

虽然可以使用 PL/pgSQL 语言以及 PostgreSQL 自己的正则表达式功能来实现与上面的示例中完全相同的验证功能，但我们刚刚还是光明正大地拿来了一段别人的成熟代码，然后没花任何时间就实现了预期的功能。对开发人员来说，这难道不是最理想的人生吗？如果你

是一名 Web 开发人员，而且需要在客户端和数据库服务器端同时对某份数据进行逻辑完全相同的合法性验证，那么使用 PL/V8 可以使你的工作事半功倍，只要在客户端写好逻辑，然后复制粘贴到数据库端就可以了。

你可以创建一张带一个 text 字段的表（该 text 字段用于存储 JavaScript 函数），然后将这些校验函数都存入该表，然后通过一些设置可以实现 PostgreSQL 启动时自动加载表中存储的这些函数，此后在数据库的任何 PL/V8 函数中都可以随便调用这些函数了。具体的操作步骤请参考 Andrew Dunstan 的博文"在 PLV8 中加载 JavaScript 模块"（http://adpgtech.blogspot.com/2013/03/loading-useful-modules-in-plv8.html）。能够实现 JavaScript 函数自动加载的关键在于 PL/V8 原生支持了 `eval` 函数，通过该函数可以动态执行任何 JavaScript 命令，因此才得以在启动阶段就对函数进行预加载。

我们通过一个在线语法转换器（js2coffee.org）将示例 8-17 中的 JavaScript 函数转换成了基于 CoffeeScript 语法的函数，如示例 8-18 所示。

示例 8-18：使用 PL/Coffee 语言编写的电子邮件地址校验函数

```
CREATE OR REPLACE FUNCTION
validate_email(email text) returns boolean as
$$
    re = /\S+@\S+\.\S+/
    return re.test email
$$
LANGUAGE plcoffee IMMUTABLE STRICT;
```

CoffeeScript 与 JavaScript 之间的语法差别并不大，主要的变化是去掉了小括号、大括号和分号。LiveScript 的语法和 CoffeeScript 的语法更是完全一样，唯一的差别就是语言声明要改为 `LANGUAGE plls`。

8.5.2 使用PL/V8来编写聚合函数

在示例 8-19 中，我们使用 PL/V8 语言重写了计算几何平均值的聚合函数（原例子请参考 8.2.2 节中的示例 8-9）。

示例 8-19：PL/V8 版的几何平均值聚合函数的状态切换函数

```
CREATE OR REPLACE FUNCTION geom_mean_state(prev numeric[2], next numeric)
RETURNS numeric[2] AS
$$
    return (next == null || next == 0) ? prev :
    [(prev[0] == null)? 0: prev[0] + Math.log(next), prev[1] + 1];
$$
LANGUAGE plv8 IMMUTABLE;
```

示例 8-20：PL/V8 版的几何平均值聚合函数的最终处理函数

```
CREATE OR REPLACE FUNCTION geom_mean_final(in_num numeric[2])
RETURNS numeric AS
```

```
$$
  return in_num[1] > 0 ? Math.exp(in_num[0]/in_num[1]) : 0;
$$
LANGUAGE plv8 IMMUTABLE;
```

`CREATE AGGREGATE` 命令将各子函数整合成我们想要的聚合函数，不管子函数采用什么语言编写，此处语法都是一样的，具体如示例 8-21 所示。

示例 8-21：PL/V8 版的几何平均值聚合函数的最终定义

```
CREATE AGGREGATE geom_mean(numeric) (
  SFUNC=geom_mean_state,
  STYPE=numeric[],
  FINALFUNC=geom_mean_final,
  INITCOND='{0,0}'
);
```

你可以把示例 8-9 再次运行一遍，但把其中的 `geom_mean` 函数换为此处的 PL/V8 版本，得到的结果与当初使用 SQL 版本 `geom_mean` 计算得到的结果肯定是一样的，但 PL/V8 版本的运算速度比 SQL 版本要快两到三倍。对于数学运算来说，你会发现在很多情况下用 PL/V8 语言编写的函数要比用 SQL 编写的相同功能的函数要快 10 到 20 倍。

第 9 章

查询性能调优

我们在使用数据库的过程中迟早会遇到语句性能问题，常见的解决方案是优化 SQL 语句本身的写法，辅以建立合适的索引以及更新规划器分析过程中所需的统计信息。为了帮助用户实现这些优化动作，PostgreSQL 提供了内置的执行计划解释器，通过它可以展示出一个 SQL 语句的执行计划。如果你了解如何编写正确的 SQL 语句，如何建立合适的索引，再加上执行计划解释器的帮助，那么写出优秀的 SQL 语句并充分利用硬件的最大计算能力应该不是难事。

9.1 通过EXPLAIN命令查看语句执行计划

要定位语句的性能问题，最简单直接的方法就是使用 EXPLAIN 和 EXPLAIN(ANALYZE) 命令来分析其执行计划。PostgreSQL 从很早的版本开始就已经支持该命令，并且历年来其功能一直在不断地演进，目前已经非常成熟，可以展示出一个语句的执行计划方方面面的细节。在演进过程中，该命令支持的输出格式也越来越丰富。从 9.0 版开始，你甚至可以将输出转储为 XML、JSON 或者 YAML 格式。

对于普通用户来说，该功能最令人激动的一次强化是几年前 pgAdmin 引入的图形化展示执行计划的能力。借助于这种能力，你只需通过仔细观察执行计划图即可了解语句的瓶颈点在哪里，哪些表应该建索引，以及实际的执行路径与预期的执行路径是否一致。

9.1.1 EXPLAIN选项

要执行非图形化的 EXPLAIN 分析，只需在 SQL 语句前加上 EXPLAIN 或者 EXPLAIN(ANALYZE)，

然后再执行即可。

EXPLAIN 本身的执行效果是输出执行计划而并不执行 SQL 语句本身，加上 ANALYZE 实参之后（就像 EXPLAIN (ANALYZE)）的执行效果是执行该 SQL 语句本身而且会将实际执行情况与执行计划进行对比分析，这可以用来评估执行计划的准确性。

在 EXPLAIN 后增加 VERBOSE 实参将使得输出的执行计划精确到列级别。还有一个必须与 ANALYZE 实参联用的 BUFFERS 实参，其语法为 EXPLAIN(ANALYZE,BUFFERS)，通过它可以显示出执行计划过程中重用缓存数据时的命中次数，这个数字越大就表示本次查询过程中从内存缓存中获取的记录数越多，这些数据是之前的查询执行过程中缓存下来的，缓存中已有的数据块就不需要再从磁盘读取了。

完整的 SQL 语句执行计划解释语法是这样的：EXPLAIN(ANALYZE,VERBOSE,BUFFERS)+ 查询语句，执行后输出的结果包括执行时间、列的输出以及缓存命中次数等。

要想使用图形化 EXPLAIN，当然必须得有比如 pgAdmin 这样的图形化界面工具。通过 pgAdmin 启动图形化 EXPLAIN 之后，请照常编辑查询，而不是执行它，然后从下拉菜单中选择 EXPLAIN 或者 EXPLAIN(ANALYZE)。有人可能对图形化界面操作不屑一顾，认为字符界面对他来说已经足够，那么我们只能说：请多保重。

对于 UPDATE 或者 INSERT 这种 DML 语句来说，如果仅希望查看其执行计划而不希望真正执行，可以把这个语句包装成一个事务块，即语句之前加 BEGIN，之后加 ROLLBACK。

9.1.2 运行示例以及输出内容解释

我们找个例子来试验一下。首先使用 EXPLAIN(ANALYZE) 命令，SQL 命令中使用的是前面的示例 4-1 和示例 4-2 中创建的表。

我们想先测试语句不使用索引的情况，因此先将表上的主键删掉。

```
ALTER TABLE census.hisp_pop DROP CONSTRAINT IF EXISTS hisp_pop_pkey;
```

这样该表上的语句就不会再使用索引，从示例 9-1 中可以看到此时的执行计划为全表扫描策略。

示例 9-1：使用 EXPLAIN (ANALYZE) 查看全表扫描的执行计划

```
EXPLAIN (ANALYZE) SELECT tract_id, hispanic_or_latino
FROM census.hisp_pop
WHERE tract_id = '25025010103';
```

示例 9-2 是示例 9-1 的执行输出结果。

示例 9-2：EXPLAIN(ANALYZE) 的执行结果

```
Seq Scan on hisp_pop
    (cost=0.00..33.48 rows=1 width=16)
    (actual time=0.205..0.339 rows=1 loops=1)
    Filter: ((tract_id)::text = '25025010103'::text)
    Rows Removed by Filter: 1477
Total runtime: 0.360 ms
```

几乎所有的执行计划都会包含多个步骤，每一步骤又可能会有若干子步骤。每一步骤会有一个估算的执行时间范围，看起来像这样：cost=0.00..33.48，如示例 9-2 所示。其中第一个数字 0.00 是估算的该步骤起始执行时间，第二个数字 33.84 是估算的该步骤总执行时间。起始执行时间点之前会执行一些后续计算的准备动作，而读取数据、索引扫描、多表数据关联整合等动作都是在起始执行时间点之后发生的。如果执行方式为全表扫描，那么其起始执行时间点为 0，因为这种场景下规划器只是简单地立即开始扫描全表数据，没有什么预备动作。

请注意，估算的执行时间（cost）的单位并不是真实的时间单位，而是取决于硬件环境以及规划器的执行时间单位常数（seq_page_cost 和 cpu_tuple_cost）。因此，cost 值仅具有相对意义，可用于比较同一台物理服务器上多个执行计划之间的效率。规划器的任务就是要选择出总体 cost 值最低的一个执行计划。

因为我们选择了在示例 9-1 中包含 ANALYZE 实参，因此规划器将运行查询，所以我们可以查看到真正的执行时间统计。

通过示例 9-2 中的执行计划可以看到规划器选择了全表扫描策略，因为没有任何索引可用。下面输出的 Rows Removed by Filter:1477 是扫描过程中排除掉的不符合条件的记录数。

在 PostgreSQL 9.4 版中，EXPLAIN 输出的执行计划中区分了分析执行计划的时间和真正的执行时间，并将二者分开单列。执行计划分析时间就是规划器分析出最终执行计划所消耗的时间；执行时间是按照执行计划执行并得到最终结果所用的时间。9.4 版的输出如示例 9-3。

示例 9-3：9.4 版中 EXPLAIN(ANALYZE) 命令的输出结果

```
Seq Scan on hisp_pop
    (cost=0.00..33.48 rows=1 width=16) (actual time=0.213..0.346 rows=1 loops=1)
    Filter: ((tract_id)::text = '25025010103'::text)
    Rows Removed by Filter: 1477
Planning time: 0.095 ms
Execution time: 0.381 ms
```

我们把主键重新建起来：

```
ALTER TABLE census.hisp_pop ADD CONSTRAINT hisp_pop_pkey PRIMARY KEY(tract_id);
```

再次执行示例 9-1 的语句，得到的输出如示例 9-4 所示（基于 PostgreSQL 9.4 版执行）。

示例 9-4：利用了索引的执行计划

```
Index Scan using idx_hisp_pop_tract_id_pat on hisp_pop
    (cost=0.28..8.29 rows=1 width=16) (actual time=0.018..0.019 rows=1 loops=1)
    Index Cond: ((tract_id)::text = '25025010103'::text)
Planning time: 0.110 ms
Execution time: 0.046 ms
```

此场景下规划器判定使用索引会比全表扫描效率更高，因此在执行计划中使用了索引扫描策略。估算的执行时间从 33.48 降为 8.29。起始执行时间点也不再是 0，因为规划器需要先扫描索引，然后才能把命中的记录从磁盘取出来（如果所需数据已经存在于内存缓存中，也有可能是直接从内存取）。你也可以看到规划器不再需要扫描 1477 条记录，这极大地降低了执行成本。

对于如示例 9-5 所示的更复杂的查询，其执行计划中会包含更多的子步骤。最终执行的步骤显示时总是排在最前，其中记录的估算时间和真实时间就是其所有子步骤相应项目之和。子步骤在显示时是按照其层级向右逐级缩进的。

示例 9-5：带 GROUP BY 和 SUM 的语句的执行计划

```
EXPLAIN (ANALYZE)
SELECT left(tract_id,5) AS county_code, SUM(white_alone) As w
FROM census.hisp_pop
WHERE tract_id BETWEEN  '25025000000' AND '25025999999'
GROUP BY county_code;
```

示例 9-6 中记录的是示例 9-5 中语句的执行计划，其中包含分组和求和操作。

示例 9-6：包含哈希聚合策略的执行计划

```
HashAggregate
    (cost=29.57..32.45 rows=192 width=16) (actual time=0.664..0.664 rows=1 loops=1)
    Group Key: "left"((tract_id)::text, 5)
    -> Bitmap Heap Scan on hisp_pop
          (cost=10.25..28.61 rows=192 width=16) (actual time=0.441..0.550 rows=204
loops=1)
        Recheck Cond:
            (((tract_id)::text >= '25025000000'::text) AND
            ((tract_id)::text <= '25025999999'::text))
        Heap Blocks: exact=15
        -> Bitmap Index Scan on hisp_pop_pkey
             (cost=0.00..10.20 rows=192 width=0) (actual time=0.421..0.421 rows=204
loops=1)
            Index Cond:
                (((tract_id)::text >= '25025000000'::text) AND
                ((tract_id)::text <= '25025999999'::text))
Planning time: 4.835 ms
Execution time: 0.732 ms
```

示例 9-6 中所示执行计划的顶层步骤是一个哈希聚合操作。该操作包含一个位图表扫描子节点，该位图表扫描子节点又包含了若干位图索引扫描子节点。在本例中，因为我们是第

一次执行此语句，所以执行计划分析时间远远超过了真正的执行时间。但 PostgreSQL 有执行计划缓存功能，所以如果我们再次执行此语句，或者执行一个可以共享缓存下来的执行计划的类似语句，那么执行计划的分析时间就会大大减少。

9.1.3 图形化展示执行计划

如果你觉得阅读纯文字形式的执行计划是一件痛苦的事，那么图 9-1 中演示的图形化执行计划会解除你的烦恼。

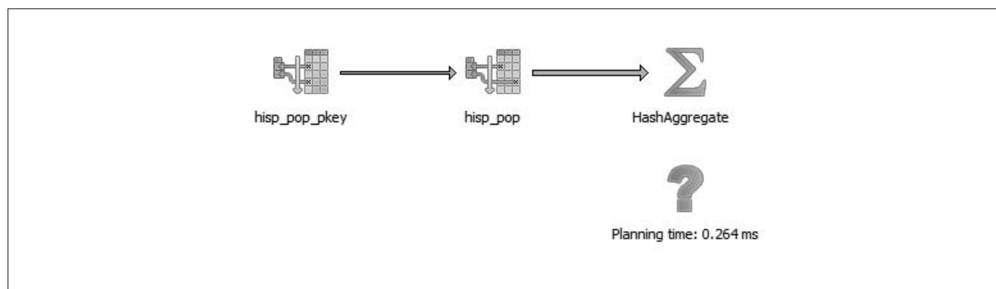


图 9-1：图形化展示执行计划

你只需将鼠标放到图标上就能看到每个步骤的详细信息。在结束本节之前，我们要向你介绍一个表格形式的执行计划展示工具（http://explain.depesz.com/），该工具由 Hubert Lubaczewski 创建，在此我们向他表示感谢。打开工具地址，然后将文本格式的执行计划复制过去，就可以得到一个格式化得非常漂亮的表格，如图 9-2 所示。

HTML TEXT STATS Did it help? Consider supporting us

Per node type stats

node type	count	sum of times	% of query
Bitmap Heap Scan	1	0.129 ms	19.4 %
Bitmap Index Scan	1	0.421 ms	63.4 %
HashAggregate	1	0.114 ms	17.2 %

Per table stats

Table name	Scan count	Total time	% of query
scan type	count	sum of times	% of table
hisp_pop	1	0.129 ms	19.4 %
Bitmap Heap Scan	1	0.129 ms	100.0 %

图 9-2：在线执行计划分析工具

在输出的 HTML 表格中，你可以看到经过格式重排的带颜色分区的执行计划表，其中会以显眼的颜色高亮显示有问题的部分，如图 9-3 所示。表格中的 exclusive 列表示当前步骤的

操作所耗时间，inclusive 列表示当前步骤及其所有子步骤的操作所耗时间。

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.114	0.664	↑ 192.0	1	1	→ HashAggregate (cost=29.57..32.45 rows=192 width=16) (actual time=0.664..0.664 rows=1 loops=1) Group Key: "left"((tract_id)::text, 5)
2.	0.129	0.550	↓ 1.1	204	1	→ Bitmap Heap Scan on hisp_pop (cost=10.25..28.61 rows=192 width=16) (actual time=0.441..0.550 rows=204 loops=1) Recheck Cond: (((tract_id)::text >= '25025000000'::text) AND ((tract_id)::text <= '25025999999'::text)) Heap Blocks: exact=15
3.	0.421	0.421	↓ 1.1	204	1	→ Bitmap Index Scan on hisp_pop_pkey (cost=0.00..10.20 rows=192 width=0) (actual time=0.421..0.421 rows=204 loops=1) Index Cond: (((tract_id)::text >= '25025000000'::text) AND ((tract_id)::text <= '25025999999'::text))

图 9-3：表格式的执行计划分析结果

尽管图 9-3 中 HTML 表格形式的执行计划提供的信息与纯文本形式的执行计划其实是一模一样的，但使用“彩色编码”和“分步骤操作时间统计”这两个功能，用户可以更轻松地找出实际执行耗时与预估耗时之间的偏差。黄色、褐色以及红色的格子是需要特别注意的。

`Rows x` 这一栏表示预估查询出的记录数，`rows` 栏显示的是实际查出的记录数。上表中显示的就是预估能返回 192 行记录，但实际仅返回 1 条。下面的索引扫描步骤总共命中了 204 条记录，但其中的 203 条其实是不符合条件的伪命中，这些记录在最后的哈希聚合步骤中复查时被揪出来了。估算的记录数不准一般是因为表的统计信息未及时更新所导致。在针对某表执行耗时较长的语句之前先对其进行一下分析是很有必要的。

9.2　搜集语句的执行统计信息

性能调优的第一步就是要确定哪些语句是性能瓶颈。PostgreSQL 提供了一个名为 `pg_stat_statements` 的性能监控扩展包以帮助用户找出耗时最长的语句。该扩展包能提供所有执行过的 SQL 语句的统计度量信息，包括哪些语句执行得最频繁以及每个语句的执行总耗时等。基于这些信息我们可以知道应将优化的重点放在哪里。

大多数 PostgreSQL 版本都自带 `pg_stat_statements` 扩展包，但启动时必须明确指定预加载其动态库，这样系统才会启动用于搜集统计数据的后台进程。设置方法如下所示。

(1) 在 postgresql.conf 配置文件中，将 `shared_preload_libraries = ''` 更改为 `shared_preload_libraries = 'pg_stat_statements'`。

(2) 在 postgresql.conf 文件的自定义选项部分，添加以下几行。

```
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

(3) 重启 postgresql 服务。

(4) 登录到每一个希望进行 SQL 语句性能统计的 database 中并执行以下语句：CREATE EXTENSION pg_stat_statements;。

该扩展包提供了以下两个关键功能。

- 一个名为 pg_stat_statements 的视图，其中可以查询到当前登录用户在各 database 中执行过的所有 SQL 语句的统计信息。
- 一个名为 pg_stat_statements_reset 的函数，该函数可以将到目前为止的语句执行统计信息全部清空，不过只有超级用户才有权限执行此函数。

示例 9-7 中的语句可以查出 postgresql_book 这个数据库中最耗时的 5 个 SQL 语句。

示例 9-7：找出特定 database 中最耗时的语句

```
SELECT
    query, calls, total_time, rows,
    100.0*shared_blks_hit/nullif(shared_blks_hit+shared_blks_read,0) AS hit_percent
FROM pg_stat_statements As s INNER JOIN pg_database As d On d.oid = s.dbid
WHERE d.datname = 'postgresql_book'
ORDER BY total_time DESC LIMIT 5;
```

9.3 人工干预规划器生成执行计划的过程

规划器生成执行计划的行为会受到多方面因素的影响，具体包括：是否有合适的索引、执行成本设置、执行策略设置以及数据如何分布等。本节中，我们将向你介绍多种可以对规划器施加人工干预的方法，通过这些方法可以得到更合理的执行计划。

9.3.1 策略设置

与某些其他数据库产品不同的是，PostgreSQL 查询规划器不接受索引提示，但你可以逐个查询或永久禁用各种策略设置（比如全表扫描、位图扫描、哈希聚合、哈希关联等都是一种执行策略，都有相应的策略设置），以阻止规划器选出某些效率较低的执行策略。PostgreSQL 官方手册中的“Planner Method Configuration”（规划器方法配置）这一节（http://www.postgresql.org/docs/current/static/runtime-config-query.html）中介绍了所有规划器优化设置。默认情况下，所有策略设置都已启用，此时规划器因为不受什么约束所以灵活性是最大的。如果你已经对要查询的数据的特点预先有了一定了解，那么就可以有针对性地禁用某些策略来优化语句的执行路径。不过请注意，即使你设置了某种策略为禁用，也并不意味着规划器就一定不会使用该策略。规划器仅将这些设置当作用户的建议来对待，最终的决定权还是在规划器。

我们有时候会将 enable_nestloop（嵌套循环）和 enable_seqscan（全表扫描）这两个设置设为“禁用”，因为这两种执行策略的效率是很低的，不到万不得已不应使用，所以通过禁用这两个设置可以告诉规划器“不到万不得已不要使用这两种策略”。你可以禁用这两种执行策略，但规划器在没有别的选择时还是可能会使用的，因为无论如何至少应该保证语句可以正常执行。如果你在执行计划中看到了全表扫描和嵌套循环这两种执行策略，那么我们建议核查一下到底是因为规划器已经找不到更好的策略所以不得不用，还是规划器选择了错误的策略。鉴别这两种情况的最好办法就是我们前面说过的，先禁用对应的策略然后再查看规划器是否还在使用这些策略，如果是，就说明是规划器不得不用；如果不是，则说明规划器之前选择了错误的策略。

9.3.2　你的索引被用到了吗

如果规划器选择了全表扫描策略，那就意味着后续执行过程中会从头到尾读取表中的每一条记录。规划器会在以下两种情况下选择全表扫描策略：一种情况是表上没有合适的索引能满足查询条件的要求；另外一种是规划器认为通过索引来查找数据的成本要高于全表扫描。如果你禁用了全表扫描策略，但规划器依然选择了这么干，这就说明表上无索引或者虽然有索引但不适用此语句的查询条件。有两个错是大家经常会犯的，一个是表上缺少必要的索引，另一个是索引建得不对而导致查询语句用不上。通过查询 pg_stat_user_indexes 和 pg_stat_user_tables 这两个视图可以很方便地得知你的索引是否被用上了，这两个视图是由 pg_stat_statements 扩展包提供的，关于该扩展包，我们已经在 9.2 节中介绍过了。

我们下面通过若干例子来说明。还是使用示例 7-18 中建过的表，我们在该表的数组列 fact_subcats 上建一个 GIN 索引，GIN 类型的索引是为数不多的能够支持数组类型的索引。语句如下：

```
CREATE INDEX idx_lu_fact_types ON census.lu_fact_types USING gin (fact_subcats);
```

我们接下来执行一个查询语句以验证所建索引是否有效，查询条件为：fact_subcats 数组列中要包含“White alone”和“Asian alone”这两个元素。虽然全表扫描策略的默认设置就是“启用”，但我们还是显式地设定一下以防万一。示例 9-8 显示了该语句的执行计划。

示例 9-8：允许规划器选择全表扫描策略

```
set enable_seqscan = true;
EXPLAIN (ANALYZE)
SELECT *
FROM census.lu_fact_types
WHERE fact_subcats && '{White alone, Black alone}'::varchar[];

Seq Scan on lu_fact_types
     (cost=0.00..2.85 rows=2 width=200) (actual time=0.066..0.076 rows=2 loops=1)
```

```
Filter: (fact_subcats && '{"White alone","Black alone"}'::character varying[]) Rows
Removed by Filter: 66
Planning time: 0.182 ms
Execution time: 0.108 ms
```

请注意，在启用全表扫描策略的情况下，规划器忽略了索引而选用了全表扫描策略。这可能是因为表的规模很小或者是因为索引不适用于本语句中的查询条件。在示例 9-9 中，我们执行的语句相同，但通过禁用全表扫描策略来强迫规划器使用了索引。

示例 9-9：禁用全表扫描策略，强行要求使用索引

```
set enable_seqscan = false;
EXPLAIN (ANALYZE)
SELECT *
FROM census.lu_fact_types
WHERE fact_subcats && '{White alone, Black alone}'::varchar[];

Bitmap  Heap  Scan on  lu_fact_types (cost=12.02..14.04 rows=2 width=200) (actual
time=0.058..0.058  rows=2   loops=1) Recheck  Cond:  (fact_subcats  &&  '{"White
alone","Black alone"}'::character varying[]) Heap Blocks: exact=1 -> Bitmap Index
Scan on idx_lu_fact_types
        (cost=0.00..12.02 rows=2 width=0) (actual time=0.048..0.048 rows=2 loops=1)
          Index Cond:  (fact_subcats && '{"White alone","Black alone"}'::character
varying[])
Planning time: 0.230 ms
Execution time: 0.119 ms
```

通过该执行计划可以看到，索引建得没问题，是可以使用的，但规划器评估后认为使用索引的执行成本要高于使用全表扫描，因此规划器选择了使用全表扫描策略。事实证明规划器的评估结果是对的，通过上面两个例子中最后的执行时间可以看到，使用索引查询的耗时要略微多于使用全表扫描的耗时。但随着表中的数据的增加，我们将会看到规划器优先选择索引查询策略。

为了与前面的例子做个对比，假设我们需要执行如下这样一个查询：

```
SELECT* FROM census.lu_fact_types WHERE 'White alone' = ANY(fact_subcats);
```

我们会看到不管将 enable_seqscan 设为启用还是禁用，规划器总是会选择执行全表扫描，因为此时索引无法满足查询条件的需要。因此，建立合适的索引并编写正确的、能用上索引的 SQL 是很重要的。保证了这些以后，后续的工作就是多试验几次，以确保生成的执行计划是最优的。

9.3.3　表的统计信息

你可能会认为规划器又神秘又强大，但无论如何规划器并不是神，它只是遵循一套设定好的算法来生成执行计划。关于规划器算法的内容细节已经远远超出本书的范畴，在此不做讨论。虽然规划器的算法严重依赖于表的统计信息，但规划器不会在每次生成执行计划之

前临时扫描所有的相关表以获取其统计信息，因为如果那么做的话任何语句的执行都将巨慢无比，完全没有执行效率可言，因此规划器会依赖预先搜集好的表统计信息。

要想规划器能够做出准确的决定，及时准确地更新表统计信息是至关重要的。如果统计信息与实际情况相差太大，规划器就很可能会常常推导出错误的执行计划，最差的情况就是错误地选择了全表扫描策略。一般来说，平均一张表只有 20% 的记录采样率，统计信息会基于这些参与采样的记录来生成。对于非常大的表来说，采样率可能更低。你可以通过设置 STATISTICS 值来修改在每一列上采样的行数。

通过查询 pg_stats 表可以了解当前的统计信息是什么样的，查询结果如示例 9-10 所示。

```
SELECT
    attname As colname,
    n_distinct,
    most_common_vals AS common_vals,
    most_common_freqs As dist_freq
FROM pg_stats
WHERE tablename = 'facts'
ORDER BY schemaname, tablename, attname;
```

示例 9-10：数据分布直方图

```
colname      | n_distinct | common_vals      | dist_freq
-------------+------------+------------------+-----------------
fact_type_id |         68 | {135,113...      | {0.0157,0.0156333,...
perc         |        985 | {0.00,...        | {0.1845,0.0579333,0.056...
tract_id     |       1478 | {25025090300...  | {0.00116667,0.00106667,0.0...
val          |       3391 | {0.000,1.000,2...| {0.2116,0.0681333,0...
yr           |          2 | {2011,2010}      | {0.748933,0.251067}
```

pg_stats 表给出了表中指定列的值域分布图，规划器会根据此信息制定相应的执行计划。系统后台会有一个进程持续不断地更新 pg_stats 表。当表中插入或者删除大量数据后，你应该手动执行 VACUUM ANALYZE 来更新表的统计信息。VACCUM 指示将已删除的记录永久性地从表中移除，ANALYZE 指示更新表的统计信息。

对于经常参与关联查询并且在 WHERE 子句频繁使用的列，应该考虑提升采样的行数。所需执行的代码如下：

```
ALTER TABLE census.facts ALTER COLUMN fact_type_id SET STATISTICS 1000;
```

9.3.4 磁盘页的随机访问成本以及磁盘驱动器的性能

另一个会影响规划器执行策略选择的设置是 random_page_cost（随机页访问成本比，简称 RPC）比率，它表示在磁盘上顺序读取和随机读取同一条记录的性能之比。一般来说，物理磁盘速度越快（一般也会越贵），该比率就会越小。RPC 的默认值是 4，该值适用于目前市面上的大多数机械硬盘。但如果使用的是固态硬盘或者 SAN 存储系统，有必要对此值

进行调整。

你可以在 database、服务器、表空间这三个级别设置 RPC 比率。如果要在服务器级别设置该比率，请直接在 postgresql.conf 文件中设置即可。如果同一台数据库服务器上使用了不同类型的硬盘，并且不同的表空间落在不同的硬盘上，那么可以在表空间级别设置 RPC 比率，语法如下所示。

```
ALTER TABLESPACE pg_default SET (random_page_cost=2);
```

有关该设置的详细信息，请参考“关于随机页访问成本比的回顾”这篇博文（http://www.databasesoup.com/2012/05/random-page-cost-revisited.html）。这篇文章建议采用以下设置。

- 高端 NAS/SAN 存储：2.5 或者 3.0
- 亚马逊 EBS 和 Heroku 云平台：2.0
- iSCSI 和其他普通 SAN 存储：6.0，但可能变化比较大，需要按照实际情况设定
- 固态硬盘：2.0 至 2.5
- NvRAM（也叫 NAND）：1.5

9.4 数据缓存机制

如果你之前执行过一个复杂且耗时较长的查询，那么后续再次执行此查询时会发现快了很多，这是因为系统的数据缓存机制发挥了作用。如果同一个查询语句按顺序多次执行，而且这些查询涉及的底层数据并没有发生变化，那么不管这些语句是被同一个用户还是多个用户执行，最终得到的结果都应该是一样的。只要内存中还有空间可用于缓存数据，那么规划器就可能会跳过生成执行计划和从磁盘读取表数据的步骤，直接从缓存中获取数据。如果语句中使用了 CTE 表达式和结果不变式函数（这类函数的运算结果不依赖外部数据，仅依赖输入的数据，也就是说固定的输入一定能得到固定的输出），那么系统会更加倾向于进行结果集缓存。

那么如何查看系统中缓存了那些数据呢？如果你的 PostgreSQL 服务器是 9.1 版或者更新的版本，可以通过安装 `pg_buffercache` 扩展包来查看。先安装该扩展包。

```
CREATE EXTENSION pg_buffercache;
```

安装完毕后，你可以查询 `pg_buffercache` 视图，如示例 9-11 所示。

示例 9-11：查看表数据是否已被缓存

```
SELECT
    C.relname,
    COUNT(CASE WHEN B.isdirty THEN 1 ELSE NULL END) As dirty_buffers,
    COUNT(*) As num_buffers
FROM
```

```
        pg_class AS C INNER JOIN
        pg_buffercache B ON C.relfilenode = B.relfilenode INNER JOIN
        pg_database D ON B.reldatabase = D.oid AND D.datname = current_database()
    WHERE C.relname IN ('facts','lu_fact_types')
    GROUP BY C.relname;
```

示例 9-11 中的语句查出了 facts 和 lu_fact_types 这两张表中被缓存的页数。需要先实际执行一个 SQL 查询语句，才会有数据被真正缓存下来，此后示例 9-11 中的语句才能有结果。我们先执行一下下面这个语句。

```
SELECT T.fact_subcats[2], COUNT(*) As num_fact
FROM census.facts As F INNER JOIN census.lu_fact_types AS T ON F.fact_type_id =
T.fact_type_id
GROUP BY T.fact_subcats[2];
```

当再次执行上述语句时，你应该可以看到至少 10% 的性能提升，并且示例 9-11 的查询可以看到类似以下的结果。

```
   relname    | dirty_buffers | num_buffers
--------------+---------------+-------------
facts         |             0 |         736
lu_fact_types |             0 |           4
```

用于缓存数据的内存大小是可指定的，该值越大，能缓存的数据就越多。postgresql.conf 中的 shared_buffers 就是用于设置此值的，但不应设得过大，否则会耗费过多时间去扫描缓存，反而降低了性能。

由于如今的物理内存已经极其廉价，因此一般不会再出现内存不够的情况。基于这一点，我们很容易就想到了可以将一些常用的表预先缓存到内存中，这样就可以提高后续访问效率。有一个名为 pg_prewarm 的扩展包可以用于实现此功能，该扩展包从 PostgreSQL 9.4 版开始成为系统自带的扩展包。pg_prewarm 会将指定的常用表预加载到缓存中，此后不管该表是首次被用户访问还是非首次访问，响应速度总是很快。关于此特性有一篇很好的文章可供你参考："关系型数据的预热机制"（http://www.depesz.com/2014/01/10/waiting-for-9-4-pg_prewarm-a-contrib-module-for-prewarming-relationd-data/）。

9.5 编写更好的SQL语句

最好也最简单的性能调优方法就是学会编写优秀的 SQL 语句。我们在大多数客户的项目中见过的 SQL 语句都写得不够好，它们未能发挥出 PostgreSQL 的真正威力。

写出的 SQL 语句很糟糕一般有两大主要原因。第一个原因是很多人会盲目地复用以前的 SQL 编写经验。例如有人曾经写过一个使用了左连接的 SQL 语句并且执行效果还不错，那么他此后不管实际情况是什么样都一直使用左连接语法，但实际上，在有更多表参与关联运算的情况下，最好是使用内连接。与其他很多编程语言不一样，SQL 语言的编写经验

不能盲目地复用。

第二个原因是人们对于最新的 SQL 语法特性一般无法及时跟进并学习了解。如果用户不及时更新自己的知识，在新版本的 PostgreSQL 上还基于旧版本的语法编写 SQL，那么新版本中引入的那些可以提升性能、简化开发的语法特性对他来说就毫无意义。

要想能够编写出高效的 SQL 是需要很多练习的。只要你编写的 SQL 语句能得到正确结果，那么这个语句就不能算错，但其性能可能很差。本节中我们将指出人们常犯的一些错误。尽管本书是关于 PostgreSQL 的，但我们给出的这些建议其实也适用于其他的关系型数据库。

9.5.1 在SELECT语句中滥用子查询

新手们常犯的一个典型错误就是容易将子查询当成一个完全独立的数据集来使用。SQL 语言有一个与传统的编程语言很不一样的地方，就是 SQL 语言中并没有很强烈的“黑盒”概念。也就是说，编写一堆互相独立的子查询并把每个子查询当作一个“黑盒”数据块来看待，只要能得到最后结果就行，而不管其他，这种思路是错误的。它事实上割裂了子查询代码块内部处理逻辑与子查询代码块外部处理逻辑之间的联系，没有将整个 SQL 语句当成一个有机的整体来处理。从多个子查询中取数据与从多个表或者视图中取数据是一样重要的，代码写得不好效率就会很低。

示例 9-12 中演示了一个滥用子查询的例子，把子查询当黑盒使用的思想就会导致这种写法。

示例 9-12：滥用子查询

```
SELECT tract_id,
    (SELECT COUNT(*) FROM census.facts As F WHERE F.tract_id = T.tract_id) As
num_facts,
    (SELECT COUNT(*)
    FROM census.lu_fact_types As Y
    WHERE Y.fact_type_id IN (
        SELECT fact_type_id
        FROM census.facts F
        WHERE F.tract_id = T.tract_id
    )
    ) As num_fact_types
FROM census.lu_tracts As T;
```

上面的 SQL 语句如果改为示例 9-13 的写法效率会更高。下面的写法合并了多个 SELECT 动作并使用了关联查询机制，不但比上面的语句更简短，速度也更快。如果表的数据量很大或者硬件性能较差，这两种写法之间的性能差异会更明显。

示例 9-13：针对滥用子查询的语句的简化改写

```
SELECT T.tract_id,
    COUNT(f.fact_type_id) As num_facts,
    COUNT(DISTINCT fact_type_id) As num_fact_types
```

```
FROM census.lu_tracts As T LEFT JOIN census.facts As F ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

图 9-4 显示的是示例 9-12 中的语句的执行计划，为了帮你免除查看字符型执行计划的痛苦，我们选择了以图形化方式展示。图 9-5 使用 http://explain.depesz.com 站点提供的工具将其执行计划转换为以 HTML 表格方式呈现，也可以提高你的查看效率。

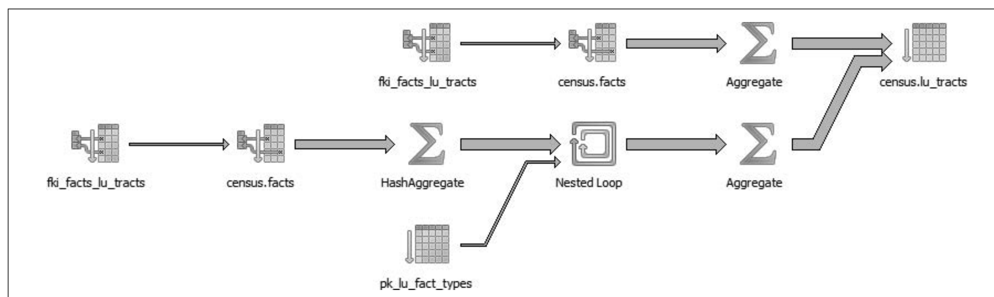


图 9-4 ：滥用子查询的 SQL 语句的执行计划图形化展示

HTML TEXT STATS

exclusive	inclusive	rows x	rows	loops	node
10.709	1292.135	↑ 1.0	1478	1	→ Seq Scan on lu_tracts t (cost=0.00..615535.37 rows=1478 width=12) (actual time
					SubPlan (forSeq Scan)
63.554	264.562	↑ 1.0	1	1478	→ Aggregate (cost=207.86..207.87 rows=1 width=0) (ac
153.712	201.008	↑ 1.0	68	1478	→ Bitmap Heap Scan on facts f (cost=4.79..207.69 rows=68 width=0) (actual time Recheck Cond: ((tract_id)::text = (t.tract_id)::text)
47.296	47.296	↑ 1.0	68	1478	→ Bitmap Index Scan on fki_facts_lu_tracts (cost=0.00..4.78 rows=68 width=0) (actual tim Index Cond: ((tract_id)::text = (t.tract_id)::text)
59.120	1016.864	↑ 1.0	1	1478	→ Aggregate (cost=208.56..208.57 rows=1 width=0) (ac
314.814	957.744	↑ 1.0	68	1478	→ Nested Loop (cost=207.86..208.39 rows=68 width=0) (actual ti
155.190	341.418	↓ 68.0	68	1478	→ HashAggregate (cost=207.86..207.87 rows=1 width=4) (actua
141.888	186.228	↑ 1.0	68	1478	→ Bitmap Heap Scan on facts f (cost=4.79..207.69 rows=68 width=4) (act Recheck Cond: ((tract_id)::text = (t.tract_id
44.340	44.340	↑ 1.0	68	1478	→ Bitmap Index Scan on fki_facts_lu_trac (cost=0.00..4.78 rows=68 width=0) (ac Index Cond: ((tract_id)::text = (t.tract_i
301.512	301.512	↑ 1.0	1	100504	→ Index Scan using pk_lu_fact_types on lu_fact (cost=0.00..0.50 rows=1 width=4) (actual time Index Cond: (fact_type_id = f.fact_type_id)

图 9-5 ：滥用子查询的 SQL 语句的执行计划表格式展示

图 9-6 显示的是示例 9-13 中的简化后语句的执行计划，从图中可以看到简化了多少执行步骤。

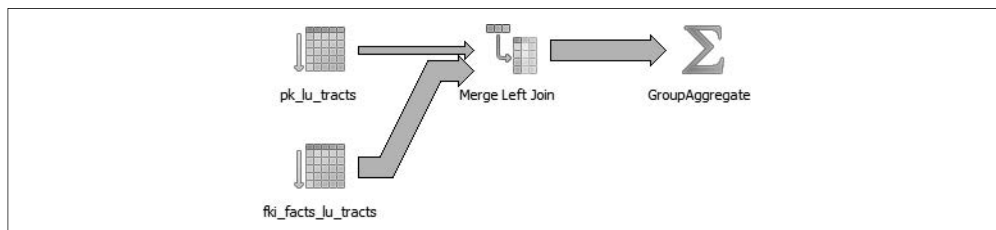


图 9-6：删除多余子查询后的执行计划图形化展示

请注意我们并没有要求你完全不用子查询，我们只是建议你在确有必要时才使用，并且使用时应当注意考虑如何将子查询与 SQL 语句的主干进行融合，也许你会发现根本不需要通过子查询来实现你所需要的功能。总之请牢记：子查询不是独立的黑盒数据块，应与主语句通盘考虑后再结合使用。

9.5.2　尽量避免使用`SELECT *`语法

`SELECT *` 经常会导致性能浪费，会出现仅仅需要 10 页数据却查出 1000 页数据这种情况，这显然会导致网络传输负担加大，而且还会出现两个你可能意想不到的问题：

第一个问题与大对象有关。PostgreSQL 会使用 TOAST（全称 The Oversized-Attribute Storage Technique，即超大尺寸属性存储技术）机制来存储二进制大对象以及超大文本。TOAST 机制会将超过主表存储限制的数据存储到一张辅助表中。因此，读取超大字段就是一个多表关联操作，而这必定是个耗时的过程。举个例子，如果某表包含文本数据，其中存储了一整部的《战争与和平》这么庞大的内容，然后你对这个表做了一个 `SELECT *` 操作，那么不难想象这个操作会慢到什么程度。

第二个问题与视图有关。我们定义视图的时候一般没办法做到完全精确地指定列，也就是说视图中一般都会带若干可能不需要的列。PostgreSQL 的视图定义功能是很强大的，你可以使用 `SELECT *` 语句来定义视图，系统会自动将星号替换为目的表的完整字段列表，你也可以在视图定义语句中包含复杂运算表达式以及关联查询。这些建视图的语句都是合法的，没有任何问题，但用户访问时就麻烦了，一旦对这种复杂视图执行 `SELECT *` 查询，那么视图定义中所有的复杂列都会经历漫长的运算过程，总体查询速度会很慢。

为了解释清楚以上观点，下面我们会在示例 9-12 中建一个带复杂子查询的 SQL 语句为基础的视图，使用的基表是 census 中的表：

```
CREATE OR REPLACE VIEW vw_stats AS
SELECT tract_id,
    (SELECT COUNT(*) FROM census.facts As F WHERE F.tract_id = T.tract_id) As
```

```
num_facts,
    (SELECT COUNT(*)
    FROM census.lu_fact_types As Y
    WHERE Y.fact_type_id IN (
        SELECT fact_type_id
        FROM census.facts F
        WHERE F.tract_id = T.tract_id
    )
    ) As num_fact_types
FROM census.lu_tracts As T;
```

如果我们针对此视图执行以下语句：

```
SELECT tract_id FROM vw_stats;
```

在我们的测试环境上该语句执行耗时大约是 21 毫秒，速度很快，因为该语句没有访问 num_facts 和 num_fact_type 这两个视图字段，这两个字段需要经过复杂的运算才能得到结果。你查看一下该语句的执行结果就可以发现其中没有任何一个步骤会访问 facts 表，因为规划器分析该语句后知道根本不需要访问此表。但如果我们使用下面的语句来访问上述视图：

```
SELECT * FROM vw_stats;
```

在我们的环境上执行时间飙升到 681 毫秒，其执行计划如图 9-4 所示。这里前后两个语句的性能差别是毫秒级，但如果表的记录数增加到千万级，列数增加到数百个，那么前后两个语句的性能差异就非常恐怖了。按照前一种写法，查询可以很快完成，你可以搞定后准点下班；按照后一种写法，你就得呆在办公室加班来等这个查询执行完毕。

9.5.3 善用CASE语法

CASE 是 ANSI SQL 标准语法，其功能其实是很强大的，但我们很少见到过有人能好好利用它，其实我们对这一点也感到很惊讶。在很多需要聚合运算的场景中，使用 CASE 语法能够有效替代子查询。接下来我们使用两个例子来演示这一点，一个使用 CASE，一个使用子查询，然后我们会比较二者的执行计划和性能差异。示例 9-14 使用了子查询语法。

示例 9-14：使用子查询而非 CASE

```
SELECT T.tract_id, COUNT(*) As tot, type_1.tot AS type_1
FROM
    census.lu_tracts AS T LEFT JOIN
    (SELECT tract_id, COUNT(*) As tot
        FROM census.facts
        WHERE fact_type_id = 131
        GROUP BY tract_id
    ) As type_1 ON T.tract_id = type_1.tract_id LEFT JOIN
    census.facts AS F ON T.tract_id = F.tract_id
GROUP BY T.tract_id, type_1.tot;
```

图 9-7 是示例 9-14 的执行计划图示。

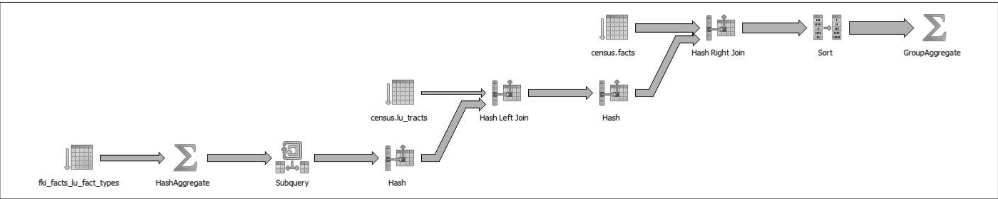


图 9-7：使用子查询而非 CASE 的执行计划

然后我们使用 CASE 语法改写这个查询。你会发现优化后的查询效率更高也更容易理解，如示例 9-15 所示。

示例 9-15：使用 CASE 语法替代子查询

```
SELECT T.tract_id, COUNT(*) As tot,
    COUNT(CASE WHEN F.fact_type_id = 131 THEN 1 ELSE NULL END) AS type_1
FROM census.lu_tracts AS T LEFT JOIN census.facts AS F
ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

图 9-8 是示例 9-15 中的语句的执行计划图示。

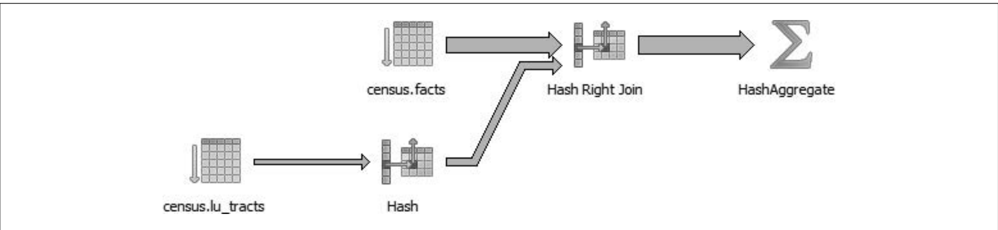


图 9-8：使用 CASE 代替子查询后的执行计划

尽管优化后的语句依然没用上 fact_type 索引，但其执行效率还是提升了，因为规划器仅对 facts 表做了一次扫描。一般来说，执行计划越短小，其执行过程就越容易理解，执行效率也越高，不过这并不是绝对的。

9.5.4 使用Filter语法替代CASE语法

PostgreSQL 9.4 版中引入了新的 FILTER 关键字，我们在 7.3 节中介绍过其用法。在使用了 CASE 的聚合函数中总是可以用 FILTER 来代替 CASE 的，替换以后不但语法上看起来更整洁而且执行效率也会有所提高。在示例 9-16 中，我们使用 FILTER 对示例 9-15 的语句进行了改写。

示例 9-16：使用 FILTER 语法来替代子查询

```
SELECT T.tract_id, COUNT(*) As tot,
    COUNT(*) FILTER(WHERE F.fact_type_id = 131) AS type_1
FROM census.lu_tracts AS T LEFT JOIN census.facts AS F
ON T.tract_id = F.tract_id
GROUP BY T.tract_id;
```

在我们的测试环境上，对示例 9-15 的语句用 FILTER 替换 CASE 后性能提升仅有大约 1 毫秒，而且两种写法的执行计划也基本类似。

第10章
复制与外部数据

PostgreSQL 有很多方法可以实现与外部数据源之间的数据共享。第一种就是 PostgreSQL 自带的复制功能，通过该功能可以在另外一台服务器上创建出当前服务器的一个镜像。第二种方法是使用第三方插件，其中许多插件可以免费使用，并且其可靠性也是久经考验的。第三种方法是使用从 9.1 版起开始支持的外部数据封装器（Foreign Data Wrapper，FDW）。FDW 支持大量的外部数据源，从 9.3 版开始，有些 FDW（比如 `postgres_fdw` 和 `hadoop_fdw`）也开始支持对外部数据的修改。

10.1　复制功能概览

很多情况下我们都需要使用数据库复制功能，但不管具体场景如何，其根本原因都可以归结为两个：提升数据的可用性和可扩展性。从可用性的角度看，如果主服务器宕机了，备用服务器应立即接管并继续提供服务。对于规模较小的数据库来说，要达到此目标只需要保证你有另一台备用的物理服务器，并将数据库恢复到该服务器上即可。但对于规模很大的数据库（数据量为 TB 级别）来说，恢复过程本身可能需要好几个小时，而且此过程中系统无法对外提供服务。为尽量减少服务中断时间，你就需要使用复制功能。我们再从可扩展性的角度来考虑这个问题，假设你开设了一个网站，做着饲养并出售象鼩的买卖。做了几年之后，你已经拥有了几千只象鼩。全世界的客户都涌到你的网站来查看并购买，由于流量过大导致网站过载并无法正常处理请求，那么这时候就需要复制功能出马了。只需设定一个只读的从属服务器作为主服务器的镜像，然后就可以把海量的读请求分流到从属服务器上，只有当买家真正下单时才需要到主服务器上执行操作。

10.1.1 复制功能涉及的术语

在我们深入探讨复制功能之前，先介绍一下相关术语。

- 主服务器（Master）

  主服务器是作为要复制数据的源头的数据库服务器，所有更新都在其上发生。使用 PostgreSQL 的内置复制功能时，仅允许使用一个主服务器。已有计划要支持多主服务器复制方案，请关注未来版本。

- 从属服务器（Slave）

  从属服务器使用复制的数据并提供主服务器的副本。尽管也有人谈到一些听起来更悦耳的术语，比如：订阅者（subscriber）、代理（agent）等，但从属服务器（slave）这个名称仍是最贴切的。PostgreSQL 内置复制目前仅支持只读从属服务器。

- 预写日志（Write Ahead Log，WAL）

  WAL 就是记录所有已完成事务信息的日志文件，在其他数据库产品中一般称为事务日志。为了支持复制功能，PostgreSQL 将主服务器的 WAL 日志向从属服务器开放，然后从属服务器持续地将这些日志取到本地，然后将其中记载的事务重演一遍，这样就实现了数据同步。

- 同步复制（Synchronous）

  在事务提交阶段，PostgreSQL 需保证已经将此事务中所做的修改成功同步到至少一个从属服务器，然后才能向用户反馈事务提交成功。这种工作模式保证了主服务器和从属服务器的数据在同一个事务内被同步修改，因此称为同步复制。如果配置了多个从属服务器，只要写入一个成功就算提交成功。

- 异步复制（Asynchronous）

  在事务提交阶段，主服务器上提交成功就算成功，不需要等待从属服务器的数据更新成功。当从属服务器位于远端时该模式就比较有用了，因为可以避免网络延迟的影响。但有利必有弊，该模式下从属服务器的数据更新不够及时，与主服务器之间会有一些延迟。当发生传输失败时，从属服务器可能会丢失一些事务数据。

- 流式复制（Streaming）

  从 PostgreSQL 9.0 版开始支持流式复制。在此前的版本中，WAL 日志是通过直接复制文件的方式从主服务器传递到从属服务器，但在流式复制模式下是通过消息来传递的。

- 级联复制（Cascading replication）

  从 9.2 版开始，一个从属服务器可以把 WAL 日志传递给另一个从属服务器，而不需要所有的从属服务器都从主服务器取 WAL 日志，这进一步减轻了主服务器的负担。这种模式下，有的从属服务器可以作为同步的数据源从而继续向别的从属服务器传播 WAL

数据，从这个角度看，其作用类似于主服务器。注意，这种扮演着“WAL 日志二传手”角色的从属服务器是只读的，它们也被称为级联从属服务器。

- 重新选主（Remastering）
  重新选主是指从所有的从属服务器中选择一个并将其身份提升为主服务器的过程。在 9.2 版及之前的版本中，如果要支持重新选主，要求复制机制必须使用基于文件的 WAL 日志传播模式，而不能使用基于流式消息的传播方式。9.3 版中解除了这个限制，引入了基于流消息复制的选主机制，该模式下选主时仅依靠流消息，而不再需要访问 WAL 日志文件，同时从属服务器也不需要经历一次重新复制过程。直到 9.4 版为止，重新选主还会要求整个数据库服务重启一次，将来的版本中可能会改进这一点。



未记录 WAL 日志的表不能被复制。

10.1.2　复制机制的演进

PostgreSQL 的复制机制依赖于 WAL 日志的传播。在 9.3 版之前，流式复制要求发送端和接收端的硬件架构必须相同，这样可以保证传送的日志流内容能够在接收端正确地解析和执行。在 9.3 版及之后的版本中，流式复制机制已经做到了与硬件架构无关，但仍然要求两端运行的 PostgreSQL 服务器软件版本相同。

PostgreSQL 内置的复制机制在以下大版本中的演进历史。

(1) 9.0 版之前，PostgreSQL 仅提供异步温从属服务器。温从属服务器是指将 WAL 日志取过来之后保持本端数据与主服务器同步，但不提供对外查询能力，仅作为一个备用服务器存在。
(2) 9.0 版引入了异步热从属服务器以及流式复制机制。异步热从属服务器允许用户针对热从属服务器执行只读查询；流式复制机制使得日志传递不再依赖文件访问而是通过数据库之间的连接来传递日志消息。
(3) 9.1 版开始支持同步复制。
(4) 9.2 版引入了对级联复制的支持，该机制主要的优点是可以降低延迟。对于从属服务器来说，从临近的另一个从属服务器取得事务日志要比从遥远的主服务器获取快得多。

10.1.3　第三方复制解决方案

除了 PostgreSQL 自带的复制机制外，还有很多第三方提供的复制工具，Slony 和 Bucardo 是其中应用最广泛的两个，而且都是开源的。尽管 PostgreSQL 原生复制机制在每个版本中

都会得到功能强化，但 Slony、Bucardo 以及其他第三方工具仍然在灵活性方面有着原生复制机制难以比拟的优势：它们支持仅复制单个 database 或者单张表；它们也不要求复制的源端和目的端的 PostgreSQL 版本和操作系统相同；它们还支持多主复制。但它们也有缺点：两个工具均依赖于新建额外的触发器来触发复制动作，因此它们对系统架构有一定的"侵入性"；并且它们一般不支持建表、安装扩展包等 DDL 操作的同步。

目前仍处于 BETA 版阶段的 PG-XC 项目已经开始吸引了一些用户的注意。PG-XC 这个项目的设计目标是实现一套分布式数据库，而不是另一套数据复制机制。其设计思想更重视的是保证系统的可扩张性而非高可用性。PG-XC 不是基于 PostgreSQL 的一套插件，而是一个完全独立的分支，致力于提供一套写能力可扩展的多主服务器对称集群，其目标非常类似于 Oracle RAC。

我们强烈建议你在决定使用哪一种具体产品之前参考一下"流行的第三方复制解决方案的对比"这篇文章（http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling）。

10.2　复制环境的搭建

本节中我们会完整介绍一遍搭建复制环境的所有步骤，以下使用 9.0 版开始支持的流式复制模式，该模式基于主服务器和从属服务器之间的数据库连接来进行 WAL 日志传输。我们还将使用 9.1 版开始支持的一个特性，该特性可以很简单地设置好专用于复制的用户账号。

10.2.1　主服务器的配置

主服务器的基本配置步骤如下所示。

(1) 创建一个专用于复制的用户账号。

```
CREATE ROLE pgrepuser REPLICATION LOGIN PASSWORD 'woohoo';
```

(2) 在 postgresql.conf 中设好以下配置项。

```
listen_addresses = *
wal_level = hot_standby
archive_mode = on
max_wal_senders = 2
wal_keep_segments = 10
```

以上设置在 PostgreSQL 官方手册的"服务器配置：复制"这一节（http://www.postgresql.org/docs/current/interactive/runtime-config-replication.html）中有详细介绍。

(3) 在 postgresql.conf 中添加 archive_command 配置指令，该指令是一个命令行字符串，表示希望将 WAL 日志保存到哪里。在流式复制模式下该配置指令中的目标路径可设定为任何路径。更多有关此配置指令的信息，请参见 PostgreSQL 官方手册中的“PostgreSQL PGStandby 工具介绍”一节（http://www.postgresql.org/docs/current/interactive/pgstandby.html）。

   在 Linux/Unix 上，archive_command 行可参照如下格式设定：

```
archive_command = 'cp %p ../archive/%f'
```

   你也可以使用 rsync 命令替代 cp 以实现异地归档：

```
archive_command = 'rsync -av %p postgres@192.168.0.10:archive/%f'
```

   在 Windows 上可按如下形式设定：

```
archive_command = 'copy %p ..\\archive\\%f'
```

(4) 在 pg_hba.conf 文件中设置一条权限规则，以允许从属服务器作为复制体系中的客户端连到主服务器。例如以下这条规则所代表的含义是：允许一个名为 pgrepuser 的角色连接到主服务器，其 IP 地址范围为 192.168.0.1 到 192.168.0.254，验证方式为基于 MD5 的加密密码。

```
host replication pgrepuser 192.168.0.0/24 md5
```

(5) 关闭 PostgreSQL 服务然后将 data 文件夹下除了 pg_xlog 和 pg_log 这两个文件夹以外的所有内容都复制到从属服务器相同的位置。请确保从属服务器 data 文件夹下存在 pg_xlog 和 pg_log 这两个文件夹，但这两个文件夹都是空的。如果你的数据库规模庞大，不允许停机并实施数据冷复制，那么可以使用 PostgreSQL 安装路径下的 bin 文件夹中的 pg_basebackup 工具，该工具可以为指定目录下的数据文件创建一份热副本，所谓“热副本”是指 postgres 服务还在运行期间就对其实施数据复制动作。

10.2.2 从属服务器的配置

建议从属服务器与主服务器的各项系统配置完全相同，这会为你减少很多麻烦，特别是当你需要搭建一套用于保障系统高可用的主备倒换环境时，这一点尤其重要。从属服务器节点必须要能够处理主服务器发来的 WAL 事务日志。具体配置步骤如下。

(1) 新建一个数据库实例作为从属服务器，要求采用与主服务器相同的 PostgreSQL 版本（最好是小版本号也完全相同），并且操作系统也相同，操作系统打的补丁也相同。事实上官方并不要求主服务器和从属服务器的软件平台完全相同，其实你可以尝试一下，看看在不一样的情况下是否能够配置成功，不过我们是不建议你在生产环境上这么做的。

(2) 关闭从属服务器的 PostgreSQL 服务。

(3) 使用从主服务器复制的 data 文件夹文件覆盖从属服务器上的相应文件。

(4) 将下面的配置设置添加到 postgresql.conf 文件中。

```
hot_standby = on
```

(5) 从属服务器的侦听端口不必与主服务器一样，因此可以选择在 postgresql.conf 中更改端口，也可以通过某些其他特定于操作系统的启动脚本进行更改，这些启动脚本会在启动之前设置 `PGPORT` 环境变量。任何启动脚本都将替代 postgresql.conf 中的设置。

(6) 在 data 文件夹下创建一个名为 recovery.conf 的新文件，内容如下（注意下面第二行中要修改为真实的 IP 地址和端口）。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.0.1 port=5432 user=pgrepuser password=woo-
hoo'
trigger_file = 'failover.now'
```

(7) 如果发现从属服务器处理事务日志的速度较慢，跟不上主服务器产生日志的速度，为避免主服务器产生积压，你可以在从属服务器上指定一个路径用于缓存暂未处理的日志。请在 recovery.conf 中添加如下一个代码行，该代码行在不同操作系统下会有所不同。

Linux/Unix 下：

```
restore_command = 'cp %p ../archive/%f'
```

Windows 下：

```
restore_command = 'copy %p ..\\archive\\%f'
```

本示例中，路径中指定的 archive 文件夹就是我们用于缓存日志的文件夹。

10.2.3 启动复制进程

一般情况下，我们建议先启动所有从属服务器再启动主服务器，如果顺序反过来，会导致主服务器已经开始修改数据并生成事务日志了，但从属服务器却还无法进行复制处理，这会导致主服务器的日志积压。如果在未启动主服务器的情况下先启动从属服务器，那么从属服务器日志中会报错，说无法连接到主服务器，但这没有关系，忽略即可。等所有从属服务器都启动完毕后，就可以启动主服务器了。

此时所有主从属服务器应该都是能访问的。主服务器的任何修改，包括安装一个扩展包或者是新建表这种对系统元数据的修改，都会被同步到从属服务器。从属服务器可对外提供查询服务。

如果希望某个从属服务器脱离当前的主从复制环境，即此后以一台独立的 PostgreSQL 服

务器身份而存在，请直接在其 data 文件夹下创建一个名为 failover.now 的空文件。从属服务器会在处理完当前接收到的最后一条事务日志后停止接收新的日志，然后将 recovery.conf 改名为 recovery.done。此时从属服务器已与主服务器彻底解除了复制关系，此后这台 PostgreSQL 服务器会作为一台独立的数据库服务器存在，其数据的初始状态就是它作为从属服务器时处理完最后一条事务日志后的状态。一旦从属服务器脱离了主从复制环境，就不可能再切换回主从复制状态了，要想切回去，必须按照前述步骤一切从零开始。

10.3 外部数据封装器

外部数据封装器（Foreign Data Wrapper，FDW）是 PostgreSQL 提供的一种用于访问外部数据源的手段，它是可扩展的，也兼容业界标准。该机制所支持的外部数据源包括 PostgreSQL 以及其他非 PostgreSQL 数据源。FDW 是在 9.1 版中引入的，其核心概念是“外部表”，这种表看起来和当前 PostgreSQL 中其他表的用法完全相同，但事实上其数据本体是存在于外部数据源中的，该数据源甚至可能存在于另外一台物理服务器上。一旦定义好了外部表，那么其定义就会在当前数据库中持久化，你就可以放心地与使用普通表一样使用它，FDW 完全屏蔽了与外部数据源之间的复杂通讯过程。可以通过 PGXN FDW 页面（http://pgxn.org/tag/fdw/）和 PGXN Foreign Data Wrapper 页面（http://pgxn.org/tag/foreign%20data%20wrapper/）查到 PostgreSQL 的 FDW 的一个 catalog。也可以通过 PostgreSQL Wiki FDW（http://wiki.postgresql.org/wiki/Foreign_data_wrappers）页面来找到 FDW 的用法示例。

当前最新版本的 PostgreSQL 安装时会默认安装两个 FDW：`file_fdw` 和 `postgres_fdw`。如果你需要自行封装某个外部数据源，那么请先到前面提供的两个站点查询一下别人是否已实现，如果没有，再自己做。如果封装成功，请记得发布出来与社区分享。

在 PostgreSQL 9.1 版和 9.2 版中，仅支持对外部表进行查询，9.3 版中引入了一组新的 API 实现了对外部表的修改，但 PostgreSQL 自带的 FDW 中只有 `postgres_fdw` 支持此特性。

本节中，我们将向你演示如何注册外部服务器、外部用户以及外部表，最后介绍如何查询外部表。我们使用的例子中都使用 SQL 命令行来创建和删除对象，你也可以通过 pgAdminIII 的图形化界面工具来实现相同的操作。

10.3.1 查询平面文件

可以使用 `file_fdw` 这个 FDW 来查询平面文件，它是以扩展包的形式存在的，因此可以通过以下 SQL 安装：

```
CREATE EXTENSION file_fdw;
```

尽管通过 `file_fdw` 可以直接读取数据库实例所在的本地服务器上的文件，但为了和别的

FDW 在语义上保持一致，还是得定义一个逻辑上的外部服务器。请执行以下命令来定义一个“伪”外部服务器：

```
CREATE SERVER my_server FOREIGN DATA WRAPPER file_fdw;
```

接下来要注册外部表。你可以将外部表置于任何一个 schema 中，但我们一般是创建一个单独的 schema 来容纳所有的外部表。

示例 10-1：定义基于分隔符格式文件的外部表

```
CREATE FOREIGN TABLE staging.devs (developer VARCHAR(150), company VARCHAR(150))
SERVER my_server
OPTIONS (format 'csv', header 'true', filename '/postgresql_book/ch10/devs.psv',
    delimiter '|', null ''
);
```

上面的示例中，尽管外部表映射到一个用竖杠作为分隔符的平面文件，但我们依然将其标识为“csv”格式。有人可能会根据 CSV（Comma Seperated Vlues）的名称认为只有用逗号作为分隔符的文件才能称为 CSV，但在 FDW 的术语体系中，CSV 文件就是以某种分隔符来区分列值的平面文件，不管这个分隔符具体是什么字符，都可以称之为 CSV。

上述定义步骤完成后，你就可以直接通过 SQL 访问外部表了：

```
SELECT * FROM staging.devs WHERE developer LIKE 'T%';
```

如果不再需要此外部表，可以删掉：

```
DROP FOREIGN TABLE staging.devs;
```

10.3.2 以不规则数组的形式查询不规范的平面文件

通常，平面文件数据源在每一行中会有许多不同的列，并且包含多个标题行和页脚行。如果平面文件起源于电子表格，那么这些种类的文件往往是普遍存在的。我们最喜欢的处理这种非结构化平面文件的平面文件 FDW 是 `file_textarray_fdw`。该包装器能处理任何种类的带分隔符的平面文件，即时每一行中的元素数量不一致也可以。该包装器将每一行作为一个文本数组（`text[]`）带进来。

问题是 `file_textarray_fdw` 不是 PostgreSQL 原生支持的扩展包，因此你必须手动编译安装它。大致过程如下：首先，你在安装 PostgreSQL 时需要附带安装系统头文件以便于后续的编译，然后从 Adunstan GitHub 这个站点（https://github.com/adunstan/file_text_array_fdw）下载 `file_textarray_fdw` 的源码。请注意：该站点上为每个 PostgreSQL 版本都准备了相应的源码，请确保选择的是正确的版本。在编译完成后，请将它以扩展包的形式安装好，该过程与我们前面介绍过的安装其他 FDW 的过程完全相同。

如果你的环境是 Linux/Unix，只要安装了 `postgresql-dev` 这个包，那么编译过程是很简单

的，但在 Windows 上就比较麻烦，所以我们已经为你编译好了安装包，你可以从以下几个链接中选择合适的版本下载。

- Windows-32 9.1 FDW（http://www.postgresonline.com/downloads/fdw_win32_91_bin.zip）
- Windows-32 9.2 FDW（http://www.postgresonline.com/downloads/fdw_win32_92_bin.zip）
- Windows-64 9.2 FDW（http://www.postgresonline.com/downloads/fdw_win64_92_bin.zip）
- Windows-32 9.3 FDW（ttp://www.postgresonline.com/downloads/fdw_win32_93_bin.zip）
- Windows-64 9.3 FDW（http://www.postgresonline.com/downloads/fdw_win64_93_bin.zip）

FDW 安装好以后，首先创建扩展包。

```
CREATE  EXTENSION file_textarray_fdw;
```

然后就跟安装其他 FDW 时一样，创建好外部服务器。

```
CREATE SERVER file_taserver FOREIGN DATA WRAPPER file_textarray_fdw;
```

然后设置外部表，可以将外部表放入任何一个你认为合适的 schema 中。在示例 10-2 中，我们将再次使用前面用过的 staging schema。

示例 10-2：创建一个基于文本数组的外部表

```
CREATE FOREIGN TABLE staging.factfinder_array (x text[])
SERVER file_taserver
OPTIONS (format 'csv', filename '/postgresql_book/ch10/
DEC_10_SF1_QTH1_with_ann.csv',
header 'false', delimiter ',', quote '"', encoding 'latin1', null ''
);
```

假设我们要处理这样一个 CSV 文件：文件中含有 8 个标题行，而列数多得我们不想数。当前述设置步骤都完成后，就可以直接查询这个文件的内容了。通过以下查询可以得到标题行的名称，这些标题行的第一个列标头为 `GEO.id`。

```
SELECT unnest(x) FROM staging.factfinder_array WHERE x[1] = 'GEO.id'
```

以下查询能查出数据的前两列。

```
SELECT x[1] As geo_id, x[2] As tract_id FROM staging.factfinder_array WHERE
x[1] ~ '[0-9]+';
```

当不再需要此外部表时，可以删除它。

```
DROP FOREIGN TABLE staging.factfinder_array;
```

10.3.3　查询其他PostgreSQL服务实例上的数据

`postgres_fdw` 可以实现从当前 PostgreSQL 实例连接到其他 PostgreSQL 实例并进行数据交

互。从 9.3 版开始，大多数的 PostgreSQL 发行包都包含了该 FDW。通过它还可以对其他 PostgreSQL 服务器上的数据进行修改操作，哪怕两边的 PostgreSQL 版本不一致也没关系。

首先也是要进行 FDW 扩展包的安装。

```
CREATE EXTENSION postgres_fdw;
```

然后创建外部服务器。

```
CREATE SERVER book_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', port '5432', dbname 'postgresql_book');
```

如果创建好外部服务器后需要更改连接选项或将其添加到外部服务器，则可以使用 ALTER SERVER 命令。比如，如果需要更改你所指向的服务器，可以执行以下代码行。

```
ALTER SERVER book_server OPTIONS (SET host 'prod');
```



对于主机、端口和 database 这几项的连接设置的更改只对新建立的会话生效，不会影响到已有的会话。原因是会话在开始时建立，此后就一直复用，而不会断开重连。

然后创建一个用户映射关系，所谓"用户映射"是指在远端服务器的某个角色和本地服务器的某个角色之间建立对应关系，这样本地角色可以用远端角色的权限来操作远端服务器上的数据。下面的示例中是将远端的某个角色映射到本地的 public 角色。

```
CREATE USER MAPPING FOR public SERVER book_server
OPTIONS (user 'role_on_foreign', password 'your_password');
```

这样任何能连到本地数据库的用户都可以连到远端数据库。注意：上述映射关系中的远端角色必须是一个已存在的角色，并且得有登录权限。

现在可以创建外部表了。该表可以映射远端表的全部或部分列。在示例 10-3 中，我们创建了一个外部表，映射到远端数据库上的 censu.facts 表。

示例 10-3：定义一个映射到远端 PostgreSQL 数据库的外部表

```
CREATE FOREIGN TABLE ft_facts (
    fact_type_id int NOT NULL, tract_id varchar(11),
    yr int, val numeric(12,3), perc numeric(6,2))
SERVER book_server OPTIONS (schema_name 'census', table_name 'facts');
```

上面的示例仅包含外部表的最基本的选项。默认情况下，映射到远端 PostgreSQL 数据库的外部表都是可更新的，当然前提是映射关系中所使用的远端数据库角色对映射的远端表要有修改权限。该 updatable 设置是一个布尔设置，可以在定义外部表或者外部服务器时进行更改。例如，如果想把外部表设定为只读，可以执行以下代码行。

```
ALTER FOREIGN TABLE ft_facts OPTIONS (ADD updatable 'false');
```

要将表设置回 updatable 状态，请执行以下代码行。

```
ALTER FOREIGN TABLE ft_facts OPTIONS (SET updatable 'true');
```

表级别上的 updatable 属性会替代外部服务器设置。

ALTER FOREIGN TABLE 语句除了可以更改 OPTIONS 之外，还可以添加或者删除列。有关该语句的详细介绍，请参见 PostgreSQL 官方手册中的 ALTER FOREIGN TABLE 这一节（http://www.postgresql.org/docs/current/static/sql-alterforeigntable.html）。

10.3.4 查询非传统数据源

长久以来，数据库界多元化的趋势有增无减，各种架构大相径庭的数据库如雨后春笋般层出不穷，要想紧密跟上业界潮流十分困难。这些异彩纷呈的数据库中，有的只是昙花一现，喧闹一阵子就消失无踪；有的是立志要将传统的关系型数据库挑落马下；更有另类者甚至看起来根本就不像是数据库。FDW 功能的引入就是 PostgreSQL 对这一百花齐放局面的应对策略之一。不管外界如何风云变幻，PostgreSQL 无需改变自身的核心功能，而是通过 FDW 搭建与这些异构数据库之间沟通的桥梁。

在下面的例子中，我们将展示如何使用 www_fdw 来查询来自 Web 服务的数据。该示例是从 www_fdw Examples 站点（https://github.com/cyga/www_fdw/wiki/Examples）借鉴而来。

PostgreSQL 发行包是不附带 www_fdw 的，因此需要自行编译安装。如果你使用的是 Linux/Unix 环境并且已安装了 postsql-dev 这个包，那么编译是很容易的。请从 https://github.com/cyga/www_fdw 这个链接下载最新版本的 www_fdw 源码。对于 Windows 平台来说，我们已经替你编译好了，下载链接如下：

- Windows-32 9.1（http://www.postgresonline.com/downloads/fdw_win32_91_bin.zip）
- Windows-64 9.3（http://www.postgresonline.com/downloads/fdw_win64_93_bin.zip）

首先安装 FDW 扩展包。

```
CREATE EXTENSION www_fdw;
```

然后创建针对 Google Web 服务的外部服务器。

```
CREATE SERVER www_fdw_server_google_search
FOREIGN DATA WRAPPER www_fdw
OPTIONS (uri 'http://ajax.googleapis.com/ajax/services/search/web?v=1.0');
```

www_fdw 默认支持 JSON 格式数据源，因此我们不需要在上述语句的 OPTIONS 修饰符中特地声明数据源格式。此外 www_fdw 还支持 XML 格式数据源。如果想了解更多 www_fdw 所支

持的形参的详细信息，请参阅其官方文档 `www_fdw`（https://github.com/cyga/www_fdw/wiki/Documentation）。请注意：每一个 FDW 都有其独特的设置，互不相同。

接下来选定至少一个本地角色来创建 FDW 用户映射关系。每个能连到本地库上的用户都应该有权限访问 Google 搜索服务器，因此我们将远端数据源的访问权限映射给本地的 `public` 角色。

```
CREATE USER MAPPING FOR public SERVER www_fdw_server_google_search;
```

然后创建外部表，如下面的示例所示。

示例 10-4：基于 Google Web 服务数据源创建一个外部表

```
CREATE FOREIGN TABLE www_fdw_google_search (
    q text,
    GsearchResultClass text,
    unescapedUrl text,
    url text,
    visibleUrl text,
    cacheUrl text,
    title text,
    content text
) SERVER www_fdw_server_google_search;
```

前面设置用户映射关系时未指定任何权限，因此需要进行一次授权动作，然后才可以访问外部表。

```
GRANT SELECT ON TABLE www_fdw_google_search TO public;
```

请注意，现在最精彩的部分来了：我们以 `New in PostgreSQL 9.4` 作为关键词进行搜索，并用正则表达式筛选掉返回结果中的 HTML 标签，语句如下所示。

```
SELECT regexp_replace(title, E'(?x)(< [^>]*? >)', '', 'g') As title
FROM www_fdw_google_search where q='New in PostgreSQL 9.4'
LIMIT 2;
```

瞧吧！我们真的得到了想要的搜索结果。

```
title
----------------------------------------------------
What's new in PostgreSQL 9.4 - PostgreSQL wiki
PostgreSQL: PostgreSQL 9.4 Beta 1 Released
(2 rows)
```

附录A

PostgreSQL的安装

A.1 Windows以及桌面Linux环境

EnterpriseDB 公司（http://www.enterprisedb.com/）为 Windows 和桌面 Linux 环境提供了安装包。我们推荐 Windows 用户使用此安装包。

基于图形化界面的安装包使用非常简单，其中还附带了 pgAdmin 以及一款辅助安装工具 StackBuilder，通过它可以为 PostgreSQL 安装一些插件和辅助工具，比如 JDBC 驱动、.NET 驱动、Ruby 驱动、phpPgAdmin 管理工具和 pgAgent 任务调度器等。

EnterpriseDB 提供了两个版本的 PostgreSQL 安装包：一个是官方开源版，也叫社区版；另一个是商业版，也叫 Advanced Plus 版。后者提供了对 Oracle 语法的兼容以及一些比社区版更强大的管理功能。这两个版本是有区别的，下载时请勿弄错。本书中我们讨论的所有内容都是针对官方开源版，而非闭源的 Postgres Plus Advanced Servers 版。不过由于二者出自同源，所以本书绝大部分内容对于后者也是适用的。



如果希望在同一台机器上免安装试用一下不同版本的 PostgreSQL，或者是希望从 USB 设备启动 PostgreSQL，那么 EnterpriseDB 公司提供了一种免安装的解决方案，具体内容请参考“在 Windows 下以免安装的方式启动 PostgreSQL”这篇文章（http://www.postgresonline.com/journal/archives/172-Starting-PostgreSQL-in-windows-without-install.html）。

A.2 CentOS、Fedora、Red Hat以及Scientific Linux

大多数 Linux/Unix 发行版会在其软件存储库中提供 PostgreSQL，但版本可能比较老。为了解决这个问题，很多人会使用逆向移植（“逆向移植”英文为 backport，是指将新版本的软件——比如数据库软件、工具软件、补丁包等——逆向移植到老版本的操作系统上，这样构建出来的版本包既能维持平台兼容性，又能提供新的软件特性，从而以最小的代价解决了老版本软件存在的问题，与全面升级操作系统平台相比，这是一种兼容性好、风险低的解决方案）的版本包，这种版本包的软件存储库中会提供较新版本的 PostgreSQL 软件。

对于具有冒险精神的 Linux 用户来说，可以从 PostgreSQL Yum 存储库（http://yum.postgresql.org/）下载最新版本的 PostgreSQL，包括开发中的版本。此存储库中不仅包含 PostgreSQL 服务器核心组件，还包含比较常用的扩展包。PostgreSQL 的开发团队负责维护这个存储库，并且会在第一时间发布补丁和版本更新。在本书写作之时，PostgreSQL Yum 存储库支持的操作系统平台包括 Fedora 14-20、Red Hat Enterprise Linux 4-6、CentOS 4-6 以及 Scientific Linux 5-6。如果使用的操作系统平台较老或者是仍需要 PostgreSQL 8.3 版这样的老版本，那么是无法使用最新的 PostgreSQL Yum 存储库的，请检查一下所使用的操作系统平台的老版本存储库，检查一下存储库仍在维护哪些内容。如果希望了解更多基于 YUM 的软件安装机制，请参考我们的 PostgresOnline 网站上关于 Yum 部分的内容（http://www.postgresonline.com/journal/categories/53-yum）。

A.3 Debian和Ubuntu

Ubuntu 一直都会在其发行版中提供最新版本的 PostgreSQL。Debian 的跟进会稍微慢一些。可以使用以下命令安装最新版本的 PostgreSQL。

```
sudo apt-get install postgresql-9.3
```

如果你需要编译 PostgreSQL 版本中未附带的扩展包，比如 PostGIS 或者 R，那么需要先安装开发库，命令如下所示。

```
sudo apt-get install postgresql-server-dev-9.3
```

如果你的操作系统存储库中未包含最新版本的 PostgreSQL，那么请访问 Apt PostgreSQL packages（https://wiki.postgresql.org/wiki/Apt）资源页面以获取最新的稳定版或者测试版，此页面上还同时提供一些其他的安装包，比如 PL/V8 和 PostGIS 等。截至本书截稿时为止，该页面中提供支持的操作系统包含 Debian 6-7 和 Ubuntu 10-14。

A.4 FreeBSD

FreeBSD 是一个常用的 PostgreSQL 平台。不过很多 FreeBSD 用户一般倾向于直接从源码编译安装 PostgreSQL 而不是使用一个现成的安装包。你可以从此链接获取到最新的适用于 FreeBSD 平台的 PostgreSQL 测试版代码：http://www.freebsd.org/ports/database.html。

A.5 Mac OS X

在 Mac 机器上安装 PostgreSQL 有很多方法：EnterpriseDB 公司提供了独立安装包；Homebrew 包管理器使用越来越广泛，并且已经吸引了一些高级 Mac 用户；KyngChaos 这个站点上提供了最新最全的开源 GIS 系统的下载；Postgres.app 是新近出现的一个应用，但很适合新手们使用；另外历史悠久的 MacPorts 和 Fink 这两个软件发布平台也还在继续提供服务。可以看到，Mac 可用的安装源很多，但我们建议 Mac 用户最好从相同的数据源下载安装包。比如，你从 KyngChaos 上下载安装了 PostgreSQL，那么最好不要到 EnterpriseDB 网站上去下载扩展包，因为可能会出现不兼容问题。详情如下所示。

- EnterpriseDB 公司（http://www.enterprisedb.com）为 Mac OS X 提供了一个非常易用的一键式 PostgreSQL 安装包，附带了 pgAdmin 管理工具。此外该公司还提供了一个名为 stack builder 的软件，通过它可以下载常用扩展包、驱动程序、编程语言扩展以及管理工具等。
- Homebrew（http://brew.sh）是 Mac OS 下的一个安装包管理器，支持包括 PostgreSQL 在内的很多软件的安装管理。Russ Brooks 提供了一篇使用 Homebrew 安装 PostgreSQL 9.1 版的安装指南（http://www.russbrooks.com/2010/11/25/install-postgresql-9-on-os-x），你也可以参照这篇文档来安装 9.1 版之后的 PostgreSQL 版本，因为安装过程几乎没有任何差异。“利用 Brew 管理器将 PostgreSQL 从 9.2 版升级到 9.3 版”（https://mattbrictson.com/postgresql-93-brew-upgrade）这篇博文提供了版本升级的操作指导。你还能在 Homebrew PostgreSQL Wiki 站点（https://wiki.postgresql.org/wiki/Homebrew）看到大量有价值的相关文章。
- 由 Heroku 开发团队提供的 PostgreSQL.app（http://postgresapp.com/）是一个免费的桌面应用安装包，号称是 Mac 平台上最方便最易用的 PostgreSQL 安装包。该安装包一般都是基于最新版本的 PostgreSQL 构建，其中还附带了常用的扩展包，比如 PostGIS、PL/Python 和 PL/V8 等。Postgres.app 作为一个独立的应用程序运行，可以按需启动和停止，非常适合用于开发，也适用于单用户场景。
- KyngChaos 站点（http://www.kyngchaos.com/software:postgres）的内容是面向 GIS 用户的，上面不但提供了最新版本的 PostgreSQL 安装包，还提供了 PostGIS、pgRouting、R 语言和 QGIS 等扩展包。

- MacPorts（http://www.macports.org/）是 Mac OS X 平台上的一个软件发布平台，支持大量的开源软件，可以实现软件包的编译、安装、升级等操作。该系统是 Mac 操作系统上最早支持 PostgreSQL 的软件发布平台。本书写作之时，该平台支持的最新版本是 PostgreSQL 9.3 版。
- Fink（http://www.finkproject.org/）是 Mac OS X 上的另一个软件发布平台，底层基于 Debian 的 apt-get 软件安装框架。本书写作之时，它提供 9.2 版的 PostgreSQL，相比其他发布平台稍显落后。

附录B

PostgreSQL自带的命令行工具

以下内容集中介绍了 PostgreSQL 的必备命令行工具，本书的正文部分已经对它们的功能进行了翔实的介绍，此处仅列出它们的帮助信息。我们希望通过提供这部分内容来为你节省一些时间，也希望能让这本书成为你更好的工作助手。

B.1 使用pg_dump进行数据库备份

pg_dump 可备份一个 database 的全部或者部分数据。支持的备份格式有：TAR 包格式、PostgreSQL 自定义压缩格式、纯文本格式以及 SQL 文本格式。纯文本格式转储的内容中含有 psql 专有命令行，因此恢复时也需要通过 psql 工具来执行此文本。SQL 文本格式转储的是仅包含标准 CREATE 和 INSERT 命令的 SQL 脚本，恢复时你可以使用 psql 或者 pgAdmin 工具来运行该脚本。示例 B-1 显示的是 pg_dump 命令的帮助信息。如果要了解 pg_dump 的全部用法，请参见 2.7.1 节。

示例 B-1：pg_dump 帮助信息

```
pg_dump --help

pg_dump可将某个database转储为文本文件或者其他格式文件。
用法：
pg_dump [选项]... [database名]

通用选项：
-f, --file=FILENAME          输出文件名或者目录名
-F, --format=c|d|t|p         输出文件格式(自定义格式、目录格式、tar包格式、纯文本)
-j, --jobs=NUM               使用这多个并行作业进行转储 ❶
-v, --verbose                详细信息模式
```

```
-Z, --compress=0-9           压缩格式的压缩级别
--lock-wait-timeout=TIMEOUT  等待表锁超时后操作失败
--help                       显示此帮助信息并退出
--version                    输出版本信息并退出

控制输出内容的选项：
-a, --data-only              仅转储数据,而不转储schema
-b, --blobs                  在转储中包含大对象
-c, --clean                  在重新创建数据库对象之前清除(删除)数据库对象
-C, --create                 包含用于在转储中创建数据库的命令
-E, --encoding=ENCODING      以ENCODING编码格式转储数据
-n, --schema=SCHEMA          仅转储命名schema
-N, --exclude-schema=SCHEMA  不转储命名schema
-o, --oids                   在转储中包含OID
-O, --no-owner               以纯文本格式跳过对象所有权的恢复
-s, --schema-only            仅转储schema,而不转储数据
-S, --superuser=NAME         要以纯文本格式使用的超级用户用户名
-t, --table=TABLE            仅转储命名表
-T, --exclude-table=TABLE    不转储命名表
-x, --no-privileges          不转储特权（grant/revoke）
--binary-upgrade             仅供升级工具使用
--column-inserts             以带有列名的INSERT命令的形式转储数据
--disable-dollar-quoting     禁用美元(符号)引号,而是使用SQL标准引号
--disable-triggers           在仅恢复数据期间禁用触发器
--exclude-table-data=TABLE   不转储命名表中的数据 ❷
--if-exists                  删除对象时使用IF EXISTS ❸
--inserts                    以INSERT命令(而非COPY命令)的形式转储数据
--no-security-labels         不转储安全标签分配
--no-synchronized-snapshots  在并行作业中不使用同步快照,通过设置该选项可以允许在
                             PostgreSQL 9.2版之前的版本上使用pg_dump -j并发转储 ❹
--no-tablespaces             不转储表空间分配
--no-unlogged-table-data     不转储不记录WAL日志的表的数据
--quote-all-identifiers      所有标识符加引号,即使不是关键字也加
--section=SECTION            转储命名部分(包括三个部分:pre-data、data以及post-data。
                             data部分包含表记录数据、大对象数据以及序列的值;post-
                             data部分包含索引、触发器、规则和约束(除了验证检查约束）
                             的定义;pre-data部分包含此外其他所有的对象定义）❺
--serializable-deferrable    等待直至转储正常运行为止
--use-set-session-authorization 使用SET SESSION AUTHORIZATION命令代替ALTER OWNER命
                                令来设置所有权
连接选项：
-d, --dbname=DBNAME      要转储的数据库 ❻
-h, --host=主机名        数据库服务器主机或套接字目录
-p, --port=端口号        数据库服务器端口号
-U, --username=名称      作为指定数据库用户连接
-w, --no-password        永远不提示输入密码
-W, --password           强制要求输入密码(应该自动发生)
--role=ROLENAME          在转储之前执行SET ROLE命令
```

❶ ❹ PostgreSQL 9.3 版中引入的新特性。

❻

❸ PostgreSQL 9.4 版中引入的新特性。

❷ ❺ PostgreSQL 9.2 版中引入的新特性。

B.2 服务器级备份工具pg_dumpall

使用 pg_dumpall 工具可以将服务器上的所有数据库备份到单个纯文本文件或者单个纯文本 SQL 文件上。该备份工具将自动备份角色和表空间等系统级对象的信息，这类信息不属于任何一个数据库。示例 B-2 列出了 pg_dumpall 的所有帮助信息。pg_dumpall 的具体用法请参考 2.7.2 节的内容。

示例 B-2：pg_dumpall 帮助信息

```
pg_dumpall --help

pg_dumpall可以将一个PostgreSQL数据库集群中的所有数据都提取到一个SQL脚本文件中。
用法：
pg_dumpall [选项]...

通用选项：
-f, --file=FILENAME         输出文件名
--lock-wait-timeout=TIMEOUT 等待表锁超时后操作失败
--help                      显示此帮助信息并退出
--version                   输出版本信息并退出

控制输出内容的选项：
-a, --data-only              仅转储数据,而不转储schema
-c, --clean                  重新创建数据库之前清除(删除)数据库
-g, --globals-only           仅转储全局对象,而不转储数据库
-o, --oids                   在转储中包含OID
-O, --no-owner               以纯文本格式跳过对象所有权的恢复
-r, --roles-only             仅转储角色,而不转储数据库和表空间
-s, --schema-only            仅转储schema,而不转储数据
-S, --superuser=NAME         要在转储中使用的超级用户用户名
-t, --tablespaces-only       仅转储表空间,而不转储数据库和角色
-x, --no-privileges          不转储特权(grant/revoke)
--binary-upgrade             仅供升级工具使用
--column-inserts             以带有列名的INSERT命令的形式转储数据
--disable-dollar-quoting     禁用美元(符号)引号,而是使用SQL标准引号
--disable-triggers           在仅恢复数据期间禁用触发器
--inserts                    以INSERT命令(而非COPY命令)的形式转储数据
--no-security-labels         不转储安全标签分配
--no-tablespaces             不转储表空间分配
--no-unlogged-table-data     不转储不记录WAL日志的表的数据
--quote-all-identifiers      所有标识符加引号,即使不是关键字也加
--use-set-session-authorization   使用SET SESSION AUTHORIZATION命令代替ALTER OWNER
                                  命令来设置所有权

连接选项：
-d, --dbname=CONNSTR     使用连接连接串连接 ❶
-h, --host=主机名        数据库服务器主机或套接字目录
-l, --database=DBNAME    替代默认数据库
-p, --port=端口号        数据库服务器端口号
-U, --username=名称      作为指定数据库用户连接
-w, --no-password        永远不提示输入密码
-W, --password           强制要求输入密码(应该自动发生)
--role=ROLENAME          在转储之前执行SET ROLE命令
```

如果未使用-f/--file,则会将SQL脚本写到标准输出中。

❶ PostgreSQL 9.3 版中引入的新特性。

B.3 database数据恢复工具pg_restore

可以使用 pg_restore 可恢复使用 pg_dump 创建备份文件，这些备份文件的格式包括 TAR 包格式、自定义压缩格式以及目录格式等。示例 B-3 是 pg_restore 命令的帮助信息。更多有关使用 pg_restore 的实例，请参见 2.7.3 节。

示例 B-3：pg_restore 帮助信息

```
pg_restore --help

pg_restore可以从pg_dump创建的存档中恢复一个PostgreSQL数据库。
用法：
  pg_restore [选项]... [文件名]

通用选项：
-d, --dbname=NAME             连接到数据库名称
-f, --file=文件名              输出文件名
-F, --format=c|d|t            备份文件格式(应该是自动的)
-l, --list                    打印存档的汇总目录
-v, --verbose                 详细信息模式
-V, --version                 输出版本信息并退出
-?, --help                    显示此帮助信息并退出

恢复控制选项：
-a, --data-only               仅恢复数据,而不恢复schema
-c, --clean                   在重新创建数据库对象之前清除(删除)数据库对象
-C, --create                  创建目标数据库
-e, --exit-on-error           恢复期间发生错误时退出,若不设定则默认为继续恢复
-I, --index=NAME              恢复命名索引
-j, --jobs=NUM                使用这多个并行作业进行恢复
-L, --use-list=FILENAME       将此文件的目录用于选择输出或对输出进行排序
-n, --schema=NAME             仅恢复此schema中的对象
-O, --no-owner                跳过对象所有权的恢复
-P, --function=NAME(args)     恢复命名函数
-s, --schema-only             仅恢复schema,而不恢复数据
-S, --superuser=NAME          用于禁用触发器的超级用户用户名
-t, --table=NAME              恢复命名表
-T, --trigger=NAME            恢复命名触发器
-x, --no-privileges           跳过访问特权(grant/revoke)的恢复
-1, --single-transaction      作为单个事务恢复
--disable-triggers            在仅恢复数据期间禁用触发器
--no-data-for-failed-tables   如果表创建失败,则不对其进行数据恢复
--no-security-labels          不恢复安全标签
--no-tablespaces              不恢复表空间分配
--section=SECTION             恢复命名部分(包括三个部分:pre-data、data以及post-data。
                              data部分包含表记录数据、大对象数据以及序列的值;post-
                              data部分包含索引、触发器、规则和约束(除了验证检查约束)的
                              定义;pre-data部分包含此外其他所有的对象定义)
```

```
❶
--use-set-session-authorization      使用SET SESSION AUTHORIZATION命令代替ALTER
                                     OWNER命令来设置所有权

连接选项：
-h, --host=主机名          数据库服务器主机或套接字目录
-p, --port=端口号          数据库服务器端口号
-U, --username=名称        作为指定数据库用户连接
-w, --no-password          永远不提示输入密码
-W, --password             强制要求输入密码(应该自动发生)
--role=ROLENAME            在恢复之前执行SET ROLE命令
```

❶ PostgreSQL 9.2 版中引入的新特性。

B.4 交互模式下的psql命令

示例 B-4 中列出了 psql 交互模式下支持的命令行。请参考 3.1 节和 3.2 节的例子，了解其使用方法。

示例 B-4：psql 交互模式下支持的命令

```
\?

通用命令
  \copyright             显示PostgreSQL使用和分发条款
  \g [文件] or ;         执行查询(并将结果发送给文件或|管道)
  \gset [PREFIX]         执行查询并将结果存储到psql变量中 ❶
  \h [名称]              关于SQL命令语法的帮助,*代表所有命令
  \q                     退出psql
  \watch [SEC]           每隔SEC秒执行一次查询 ❷
查询缓冲区相关命令
  \e [FILE] [LINE]       使用外部编辑器编辑查询缓冲区(或文件)
  \ef [FUNCNAME [LINE]]  使用外部编辑器编辑函数定义
  \p                     显示查询缓冲区的内容
  \r                     重置(清除)查询缓冲区
  \w 文件                将查询缓冲区写入到文件
输入/输出相关命令
  \copy ...              执行SQL COPY,将数据流发送到客户端主机
  \echo [字符串]         将字符串写到标准输出
  \i 文件                从文件执行命令
  \ir FILE               与 \i类似,但是在脚本中执行时,认为目标文件的位置是当前脚本所
                         在的目录 ❸
  \o [文件]              将所有查询结果发送到文件或|管道
  \qecho [字符串]        将字符串写入到查询输出流,该命令等效于\echo,区别是所有输出将
                         写入由 \o设置的输出通道
信息查询命令
  (选项:S = 显示系统对象,+ = 附加的详细信息)
  \d[S+]                 输出表、视图和序列列表
  \d[S+]  名称           描述表、视图、序列或索引
  \da[S]  [模式]         输出聚合函数列表
  \db[+]  [模式]         输出表空间列表
  \dc[S]  [模式]         输出编码转换(conversion)列表
```

```
  \dC     [模式]        输出类型强制转换(cast)列表
  \dd[S]  [模式]        显示对象上的注释
  \ddp    [模式]        输出默认权限列表
  \dD[S]  [模式]        输出域列表
  \det[+] [模式]        输出外部表列表
  \des[+] [模式]        输出外部服务器列表
  \deu[+] [模式]        输出用户映射列表
  \dew[+] [模式]        输出外部数据封装器列表
  \df[antw][S+] [模式] 输出特定类型函数(仅a-聚合函数/n-常规函数/t-触发器函数/w-窗口
                        函数)列表
  \dF[+]  [模式]        输出文本搜索配置列表
  \dFd[+] [模式]        输出文本搜索字典列表
  \dFp[+] [模式]        输出文本搜索解析器列表
  \dFt[+] [模式]        输出文本搜索模版列表
  \dg[+]  [模式]        输出角色列表
  \di[S+] [模式]        输出索引列表
  \dl                   输出大对象列表,与\lo_list相同
  \dL[S+] [模式]        输出过程语言列表
  \dm[S+] [模式]        输出物化视图列表 ❹
  \dn[S+] [模式]        输出schema列表
  \do[S]  [模式]        输出运算符列表
  \dO[S+] [模式]        输出排序规则列表
  \dp     [模式]        输出表、视图和序列访问权限列表
  \drds [模式1 [模式2]] 输出每个database的角色设置列表
  \ds[S+] [模式]        输出序列列表
  \dt[S+] [模式]        输出表列表
  \dT[S+] [模式]        输出数据类型列表
  \du[+]  [模式]        输出角色列表
  \dv[S+] [模式]        输出视图列表
  \dE[S+] [模式]        输出外部表列表
  \dx[+]  [模式]        输出扩展列表
  \dy     [模式]        输出事件触发器列表 ❺
  \l[+]                 输出数据库列表
  \sf[+] FUNCNAME       显示函数定义
  \z      [模式]        和\dp的功能相同
格式化相关命令
  \a                    在非对齐输出模式和对齐输出模式之间切换
  \C [字符串]           设置表标题;或者如果没有,则不设置
  \f [字符串]           显示或设置非对齐查询输出的字段分隔符
  \H                    切换HTML输出模式(当前关闭)
  \pset NAME [VALUE]    设置表输出选项(NAME的可选项有format、border、expanded、
                        fieldsep、fieldsep_zero、footer、null、numericlocale、recordsep、
                        tuples_only、title、tableattr、pager)                ❻ ❼
  \t [on|off]           仅显示行(当前关闭)
  \T [字符串]           设置HTML
  \x [on|off]           切换扩展输出(当前关闭)

连接相关命令
  \c[onnect] [DBNAME|- USER|- HOST|- PORT|-] 连接到新的database(当前是“postgres”)
  \encoding [编码名称]   显示或设置客户端编码
  \password [USERNAME]   安全地为用户更改密码
  \conninfo              显示当前连接的相关信息
操作系统相关命令
  \cd [目录]             更改当前工作目录
  \setenv NAME [VALUE]   设置或取消设置环境变量 ❽
```

```
\timing [on|off]        切换命令计时开关(当前关闭)
! [命令]                在shell中执行命令或启动交互式shell
```

❶ ❷ PostgreSQL 9.3 版中引入的新特性。

❹ ❺

❸ ❻ PostgreSQL 9.2 版中引入的新特性。

❽

❼ PostgreSQL 9.4 版中引入的新特性。你可以使用不带任何实参的 `\pset` 命令来输出所有的可选项及其当前值。

B.5 非交互模式下的psql命令

示例 B-5 是非交互模式下 psql 的命令行帮助信息。关于该模式下的具体使用例子请参考 3.2 节的内容。

示例 B-5：psql 基本帮助信息

```
psql --help

psql是PostgreSQL的交互式终端。
使用方法：
psql [选项]... [database名称 [用户名]]

通用选项：
-c,--command=命令         仅运行单个命令(SQL或内部命令),然后退出
-d, --dbname=数据库名称   要连接到的数据库名称
-f, --file=文件名         从文件执行命令,然后退出
-l, --list                列出可用的数据库,然后退出
-v, --set=, --variable=名称=值 将psql变量NAME设置为VALUE
-X, --no-psqlrc           不读取启动文件(~/.psqlrc)
-1 ("one"), --single-transaction 将命令文件作为单一事务执行
--help                    显示此帮助信息并退出
--version                 输出版本信息并退出

输入和输出选项：
-a, --echo-all            回显所有来自于脚本的输入
-e, --echo-queries        回显发送给服务器的命令
-E, --echo-hidden         显示内部命令生成的查询
-L, --log-file=文件名     将会话日志发送给文件
-n, --no-readline         禁用增强命令行编辑功能(readline)
-o, --output=FILENAME     将查询结果发送给文件(或|管道)
-q, --quiet               以静默模式运行(不显示消息,仅显示查询输出)
-s, --single-step         单步模式 (每个查询均需确认)
-S, --single-line         单行模式(SQL命令不允许跨行)


输出格式选项：
-A, --no-align            非对齐表输出模式
-F, --field-separator=字符串 设置字段分隔符(默认为“|”)
```

```
-H, --html                HTML表输出模式
-P, --pset=VAR[=ARG]      将打印选项VAR设置为ARG(参见\pset命令)
-R, --record-separator=字符串 设置记录分隔符(默认为换行符)
-t, --tuples-only         仅打印行
-T, --table-attr=文本     设置HTML表标记属性(例如:宽度、边框等)
-x, --expanded            打开扩展表输出
-z, --field-separator-zero ❶
                            将字段分隔符设置为零字节
-0, --record-separator-zero ❷
                            将记录分隔符设置为零字节

连接选项:
-h, --host=主机名         数据库服务器主机或套接字目录
-p, --port=端口           数据库服务器端口(默认为"5432")
-U, --username=用户名     数据库用户名
-w, --no-password         永远不提示输入密码
-W, --password            强制要求输入密码(应该自动发生)

如需了解更多信息,请在psql中输入"\?"(用于内部命令)或者"\help"(用于SQL命令),或者参考
PostgreSQL官方手册中的psql部分。
```

❶ ❷ PostgreSQL 9.2 版中引入的新特性。

作者简介

Regina Obe 是位于美国波士顿的数据库咨询服务公司 Paragon Corporation 的负责人之一。她有着 15 年以上的数据库领域从业经验，精通多种编程语言和数据库系统，特别是在空间数据库方面尤为专长。她是 PostGIS 指导委员会成员，同时也是 PostGIS 核心开发团队的成员。 Regina 拥有麻省理工学院机械工程学士学位，是 *PostGIS in Action* 一书的作者之一。

Leo Hsu 也是 Paragon Corporation 公司的负责人之一。他有着超过 15 年的数据库领域从业经验，曾为许多不同规模的公司和组织做过数据库开发工作，对数据库领域有着非常深入的思考和研究。Leo 拥有斯坦福大学经济系统工程硕士学位以及麻省理工学院机械工程与经济学硕士学位，他也是 *PostGIS in Action* 一书的作者之一。

封面介绍

本书封面上的动物是象鼩（拉丁名为 Macroscelides proboscideus），这是一种原产于非洲的食虫性哺乳动物，广泛分布于非洲南部，因有着类似大象的长鼻而得名。它们能够适应各种各样的生存环境：无论是纳米布沙漠，还是砾石覆盖的南部非洲地区，甚至茂密的森林地带，都是它们的栖居之地。

象鼩是一种体型很小的四足动物。由于尾巴非常相似，因此象鼩外表上看起来像是老鼠或者负鼠。相较于其体型来说，它们的腿可以说是相当之长，因此它们可以跳跃行走，看起来和兔子很相似。它们的鼻子部分根据亚种的不同而长度各异，但在寻找食物时都可以左右扭动。

虽然象鼩是一种活跃的昼行性动物，但由于其个性机警，所以一般很难见到或者捕捉到它们。它们很善于伪装，在遇到危险时会迅速逃避。

象鼩并非高度群居性的动物，很多个体都是以一夫一妻的方式结伴生活并共同保护它们的领地。雌性象鼩有着类似人类女性的月经周期，其发情期会持续好几天。雌性个体怀孕以后，其妊娠期会持续 45 至 60 天，一年会生育若干胎，每胎约有 1~3 只幼鼩。幼鼩出生时其身体已经发育得比较完全，几天后就会离开巢穴。

出生五天之后，幼鼩开始进食昆虫，这些昆虫由它们的母亲捕获并衔在口中携带回来。幼鼩会在出生之后大约 15 天开始尝试独立生活并逐步减少对母亲的依赖。随后它们会圈定自己的领地，并在 41 至 46 天之内达到性成熟状态。

成年象鼩主要以无脊椎动物为食，比如昆虫、蜘蛛、蜈蚣、千足虫以及蚯蚓等。要想吃掉个头更大些的猎物对它们来说会有点困难，它们必须用脚拖住猎物，再用牙齿把食物撕扯成碎片，等这些碎片落到地上之后象鼩会像食蚁兽那样用舌头将这些碎片舔进嘴里。象鼩同时也是植食性动物，如果能找得到的话，嫩叶、种子、小型果实等也都是它们的美食。

很多出现在 O'Reilly 图书封面上的动物都濒临灭绝，它们的存在对于维持地球的物种多样性非常重要，如果你希望能为保护它们尽一份力量，请访问 animals.oreilly.com 以了解详情。

封面图片来自于 *Meyers Kleines* 词典。

欢迎加入

图灵社区 iTuring.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为"敏捷出版"，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书"出版即过时"的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用"合集"功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

TURING 图灵教育

图灵最新重点图书



大数据权威著作全新升级版！

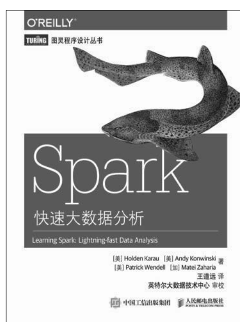
第 1 版畅销 40000 册！

本书源自作者在斯坦福大学教授的“海量数据挖掘”（CS246: Mining Massive Datasets）课程，第 1 版上市以来受到读者广泛欢迎和认可。本书以大数据环境下的数据挖掘和机器学习为重点，全面介绍了实践中行之有效的数据处理算法，是在校学生和相关从业人员的必备读物。

大数据（第 2 版）
书号：978-7-115-39525-2
定价：79.00 元



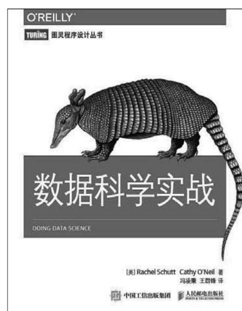
Spark 高级数据分析
书号：978-7-115-40474-9
定价：59.00 元



Spark 快速大数据分析
书号：978-7-115-40309-4
定价：59.00 元



命令行中的数据科学
书号：978-7-115-39168-1
定价：49.00 元



数据科学实战
书号：978-7-115-38349-5
定价：79.00 元



云数据中心网络技术
书号：978-7-115-40518-0
定价：49.00 元



学习 JavaScript 数据结构与算法
书号：978-7-115-40414-5
定价：39.00 元

关注图灵教育 关注图灵社区

iTuring.cn

在线出版　电子书《码农》杂志　图灵访谈……



QQ联系我们

图灵读者官方群I：218139230
图灵读者官方群II：164939616



微博联系我们

官方账号：@图灵教育 @图灵社区 @图灵新知
市场合作：@图灵袁野
写作本版书：@图灵小花 @图灵张霞
翻译英文书：@朱巍ituring @楼伟珊
翻译日文书或文章：@图灵乐馨
翻译韩文书：@图灵陈曦
电子书合作：@hi_jeanne
图灵访谈/《码农》杂志：@李盼ituring
加入我们：@王子是好人



微信联系我们



图灵教育
turingbooks



图灵访谈
ituring_interview

O'REILLY®

PostgreSQL即学即用（第2版）

你是否正在考虑将业务系统数据库迁移到PostgreSQL上？本书内容简明扼要、提纲挈领，是极佳的PostgreSQL快速上手指南，可以帮助你快速地学习、理解并运用好这款开源数据库。从本书中，你不仅能学到PostgreSQL 9.2、9.3和9.4版中的企业级特性，还将了解到PostgreSQL不但是一套数据库系统，更是一个功能强大的应用开发平台。

本书以众多示例贯穿始终，演示了如何实现在别的数据库中难以实现或者根本不可能实现的任务。本书第2版涵盖了LATERAL横向关联查询语法、增强的JSON支持、物化视图机制以及其他重要功能特性。即使你已经是PostgreSQL用户，也能从本书中学到以前未曾了解过的一些功能。

通过阅读本书，你将学到：

- **如何执行基本的数据库管理任务，比如角色管理、数据库创建以及数据备份和恢复等；**
- **如何使用psql命令行工具以及pgAdmin图形化管理工具；**
- **PostgreSQL的表、约束和索引等数据库对象的特性和使用方法；**
- **PostgreSQL所特有的若干功能强大的SQL语法；**
- **如何使用多种不同的编程语言来编写PostgreSQL函数；**
- **如何实施语句调优以充分挖掘服务器硬件的潜能；**
- **如何通过外部数据封装器来查询多种多样的外部数据；**
- **如何使用系统内置的复制筛选器进行数据复制。**

“这本书篇幅不长……对系统管理员、数据库管理员以及开发人员来说，如果有其他数据库系统的使用经验，希望不必深入研究各类细节就能够对PostgreSQL快速上手，那么阅读这本书再合适不过了。”

——Andrew Dunstan
PostgreSQL Experts公司高级顾问，
PostgreSQL核心代码提交者

Regina Obe是数据库咨询公司Paragon的负责人之一，在编程语言和数据库系统方面有15年以上的专业经验。她是PostGIS指导委员会成员，也是PostGIS核心开发团队的成员。*PostGIS in Action*一书的合著者。

Leo Hsu也是Paragon公司的负责人之一，曾为大大小小的组织开发过数据库，有15年以上的专业经验。*PostGIS in Action*一书的合著者。

PROGRAMMING/SQL

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/数据库

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-41128-0
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks